Large-Scale System Configuration with LCFG and SmartFrog

Paul Anderson <dcspaul@inf.ed.ac.uk>
George Beckett <g.beckett@epcc.ed.ac.uk>
Kostas Kavoussanakis <k.kavoussanakis@epcc.ed.ac.uk>
Guillaume Mecheneau <guillaume.mecheneau@hp.com>
Jim Paterson <jpaters2@inf.ed.ac.uk>
Peter Toft <peter.toft@hp.com>

Abstract

This is the third report from the GridWeaver project. It describes how the LCFG and SmartFrog technologies have been combined, in prototype form, to create a large-scale configuration system. First, we focus on the configuration aspects that must be modelled and examine how these are represented in the SmartFrog 2 language. Secondly, we discuss how the LCFG and SmartFrog systems have been combined to create a dynamic configuration deployment environment.

Revision 1.0 – June 06, 2003

Contents

Chapter 1

Introduction

This report is the third in a series of deliverables from the *GridWeaver* project [Edi], a collaboration between the School of Informatics at the University of Edinburgh, HP Labs in Bristol, and EPCC, as part of the UK e-Science Core Programme. GridWeaver is focused on the challenges of managing the configuration of very large-scale computational infrastructures, especially those that form parts of computing Grids. Our goal is to develop approaches and technologies that are capable of representing and realising complex system configurations of hardware, software and services, across very large-scale infrastructures incorporating high degrees of heterogeneity, dynamism and devolved control.

In our first report, *Technologies for Large Scale System Configuration Management* [ABK+02], we discussed the basic approaches in use for configuring and managing large fabrics, and examined a number of specific system configuration technologies. We concluded that there is no complete tool to tackle all aspects of fabric management. The technologies available from the consortium were known to address parts of the problem, and were found to be complementary and thus have a lot of potential.

In the second report, *Experiences and Challenges of Large-Scale System Configuration* [ABK+03], we identified the fabric configuration requirements of the emerging Grid world. We first presented a number of real world case studies to illustrate how several different organizations manage large-scale system fabrics. We examined how well the technologies employed are currently meeting the requirements placed upon them, and looked for any limits being encountered now or in the future, extending to an investigation of the fabric manager's thinking. We also introduced some Projected Use Cases that aim to predict the way in which large-scale system configuration will need to evolve in order to meet the challenges of tomorrow's fabrics. We concluded that there is a paradigm shift taking place in the way we think about computing infrastructure, and indicated the research areas that the GridWeaver project will focus on.

In the third report, we explain and model a number of interesting concepts from system configuration. We also discuss how to combine the LCFG and SmartFrog technologies in order to develop a configuration system that addresses some of these research areas.

In Chapter 2 we discuss how to represent system configurations in a way that handles the diversity, complexity and scale of the systems we are targeting. Both LCFG and SmartFrog have adopted a language-based approach to representation. Our experiences with

these technologies provide guidance as to what elements and concepts we need to be able to represent in our configurations. We then explore the suitability of HP's experimental, second generation SmartFrog description language (SF2) for representing the type of system configurations that we require, and we illustrate its use through a number of examples.

In Chapter 3 we turn our attention to how configurations are deployed—i.e. how we move from the description of a desired configuration into its realization on the fabric. We discuss how to combine the LCFG and SmartFrog technologies to create a deployment system with a natural separation of responsibilities between the technologies. LCFG focuses on the configuration of each node separately, while SmartFrog focuses on configuration aspects that span nodes with inter-nodal dependencies, and on highly dynamic aspects of configuration. The combination is achieved using adaptors written to allow LCFG to start SmartFrog and provide it with configuration data, and adaptors that allow SmartFrog to interface directly with LCFG configuration components.

The report concludes with two informational appendices, one describing the SF2 language, the second presenting more detail about the system modelling that has been done using the language.

Acknowledgements

The report is a deliverable of the *GridWeaver* project, a collaboration between the University of Edinburgh School of Informatics, HP Labs in Bristol, and EPCC.

GridWeaver is part of the UK e-Science Core Programme, and we gratefully acknowledge the programme's assistance with this activity. We also wish to thank the other members of the GridWeaver team – Carwyn Edwards, Patrick Goldsack, John Hawkins – for their contributions to the project as well as Alexander Holt for his valuable input.

For administrative purposes, GridWeaver is also known under the project name *HPFabMan*.

Chapter 2

Configuration Modelling and Description

This chapter presents the configuration modelling work undertaken as part of the GridWeaver project. The work has investigated the use of the SmartFrog[1] 2 (SF2) declarative modelling language to describe fabric configuration. There are many requirements for a general purpose configuration modelling language. These requirements have been investigated in some detail in the earlier GridWeaver report [ABK+03]. The following list gives a brief summary:

- Storage of configuration data. This is an abstraction of data that is currently stored in a variety of places: configuration file, RDMS tables, LDAP databases etc.
- Representation of the nodes on the network.
- Representation of clusters of nodes on the network.
- Representation of services and other higher level concepts on the network. A service could be provided by a simple daemon (ftpd) or by a distributed application spanning many nodes.
- Specification of invariants and constraints that apply to the fabric definition. These can be used to check consistency conditions and flag potential problems.
- Specification of dependences between elements. These could be used for impact assessment and "what-if" analysis of planned configuration changes. These can also be used as consistency checks.
- The process of moving a fabric from one configuration version to another.

The wide range of items on the list above has lead to suggestions that no single language will address the full problem space. Instead, a full solution may comprise several interdependent languages eg, a data description language, a constraint language and a procedural language to express temporal changes. This is a powerful concept. It allows a modular

[1] Smart Framework For Object Groups

approach to configuration languages and means that initial research does not have to attempt a solution to the entire problem space. In this spirit, the modelling work described in this report has focused on modelling configuration data; the relationships and other dependences within this data; and the constraints this data must satisfy to be valid. The following aspects have not been addressed:

- Modelling temporal changes to a fabric (eg. the transition from one version to another).
- Detailed design and architecture of the framework to support the modelling language (a framework has been used for the demonstrator described in Chapter 3 but should be treated as a prototype for demonstration purposes only).
- Modelling constraint satisfaction (the ability to specify constraints like *I require a web server with characteristics X* and have the configuration system find a network configuration that satisfies the constraints). In the models presented here, constraints are used to validate configurations, not to determine them. [CG99] illustrates the concepts of constraint satisfaction in system configuration.

The modelling work has focused on the SF2 language. This is a prototype-based language developed by HP Laboratories in Bristol. In this context, prototype-based languages are ones that support *instance inheritance* i.e. the ability to define a specific configuration instance and then derive other instances from this, which add or modify configuration details. These concepts have been used in a variety of other configuration languages and frameworks (Arusha [HP01], PAN [CP02] and LCFG [AS02]). Such languages appear to be ideally suited to configuration problems, as they support progressive modifications to base configuration definitions and offer the ability to compose a configuration description from a variety of smaller definitions.

SF2 is an evolution from the earlier SmartFrog 1 language that HP Laboratories developed and used within several of their products [Ser]. SmartFrog 2 has grown out of the experiences from SmartFrog 1 and is proposed as a candidate language for general system configuration problems. Some of the detailed semantics of the language have not yet been fully defined. The main purpose of the work described here was to assess the applicability of the language, as it currently stands, by applying it to real-world configuration problems. This has helped to identify weakness in the current version of the language and assist in finalising its semantics and notation.

The language evaluation has been two pronged:

- Modelling Exercises: These are paper exercises to build configuration models for selected problems. Language weaknesses will be highlighted if aspects of the configuration are found that cannot be expressed within the language semantics.
- A Demonstrator: A working prototype has been implemented to show the use of the language when applied to one specific problem.

This chapter presents the findings from the first of these evaluation approaches. Three main examples were chosen for the modelling investigations. These were:

- The configuration of OGSA applications [FKNT, TCF+] and the Globus Toolkit Version 3 [gt3].
- The configuration of a representative set of examples from LCFG.
- The configuration of printing services.

This chapter continues with a discussion of general configuration modelling issues that we found during the investigations and also lists the solutions that are proposed for these within the integrated SF-LCFG framework. We also discuss three examples of configuration descriptions, focusing on the modeling challenges each of them presents.

2.1 Key Concepts and Idioms

The LCFG framework has been in production use for about ten years, managing a medium-sized fabric of workstations with highly diverse configurations. This has highlighted a number of fundamental issues in the modelling and specification of fabric configurations. LCFG has evolved adequate (though rarely perfect) solutions to some of these problems. In this section, we discuss these solutions.

2.1.1 Aspect Composition

The specification of a complex fabric configuration covers many different aspects. For example, networking, security, laptops, machines running a particular OS, student lab machines, web servers, etc, etc. In any sufficiently large organisation, different people, or even different organisations, will be responsible for these different aspects. It is essential that we be able to author and manage these aspects independently while being able to compose them to form a valid specification for every node in the fabric.

It is important to note that these aspects frequently overlap, and the configuration system is responsible for mediating any conflicts between overlapping aspects. LCFG currently provides a *mutation* operation (see [And01]) which allows one to compose conflicting aspects using an arbitrary function. In practise however, a simple *override* function is often used, which gives priority to the last encountered value. This is far from ideal, since specifications become very sensitive to ordering, but there is often no obvious alternative.

The best way of managing aspect composition is an important open question. It seems possible that the explicit nature of the current language means that aspects are often over-specified, leaving little room for the configuration system to derive final values which are compatible with all the aspects. More use of constraints (see Section 2.1.4) may present a way of introducing more flexibility and avoiding conflicts.

Fully devolved management of aspects should also support a better security model which provides fine-grain access control to the configuration resources. LCFG does not currently support this.

2.1.2 Building a List

Apart from the overriding of simple values, one other function dominates the currently used aspect composition functions. This is the addition of some element to a list defined by some other aspect. For example, a “laptop” aspect might add some additional software packages to the list of software packages defined for the basic workstation, or a “web server” might add some additional service to a list of services to be started at boot time.

The operation of appending elements to a list is very common in LCFG and is supported by a mutation macro. Likewise, removal of items is also straightforward. However, the biggest problem in practise is again concerned with ordering. The order of the elements in the list (and even the contents, if items are removed) depends on the order in which the aspects are evaluated. Specification of declarative constraints, rather than procedural list

operations would help to solve this problem. The current LCFG compiler has experimental code for ordering some specific lists by performing a topological sort, based on a set of partial ordering constraints.

There is also a problem of naming list items so that they may be referenced by aspects which need to remove, or reorder them.

2.1.3 Spanning Maps

In many cases, the configuration of a node depends only on its own node-specific information, and a number of shared aspects – it does not depend on the configuration of other nodes. However, there are some cases where a node configuration requires information from other nodes; for example, a DHCP server may require the MAC addresses of every node which it is intended to serve. These addresses could be specified manually in the server configuration, but this would be a duplication of data, and a source of errors – it would be much better if the addresses were obtained directly from the configurations of the client nodes. LCFG provides a construction called a *spanning map* to support this type of dependency. However the LCFG spanning map is very specific, and it seems that a more general construct for explicitly representing relationships between node configurations might be useful.

It also seems possible that a more autonomic, peer-to-peer deployment model might remove some of the need for static spanning maps; for example, the DHCP server might use a runtime location protocol to locate the clients dynamically, rather than having their addresses hardwired into the static configuration.

2.1.4 Constraint Satisfaction

In LCFG, all configuration resources are currently specified explicitly. Constraints may be added, but they are only used to validate, rather than generate resource values[2]. This has two problems:

Firstly, configuration authors are forced to define explicit values for resources, even though this is not necessary. For example, it may be necessary to specify:

(i) Nodes X and Y should be servers.

When all that is really needed is:

(ii) There should be two servers on this network segment.

The over-explicitness means that unnecessary conflicts are likely to arise which require manual intervention.

[2]There is one exception to this where the compiler generates some list orders specifically from a set of ordering constraints

Secondly, looser specifications are required to support more autonomic systems which are capable of re-configuring in reaction to failures; for example, if node X fails, then there is no way of satisfying (i) above and manual intervention will be needed to reconfigure the system. Using (ii), the deployment system should be able to allocate a different server and still satisfy the configuration without any intervention.

2.1.5 Late Binding

One original philosophy of LCFG (and many similar systems) has been the use of a single central repository for all configuration information. In theory, this allows all changes to be pre-validated, and makes "what-if" analysis possible. However there will always be some configuration details which are only available on the client, at runtime; for example, if DHCP is being used, the client IP address is not available to the static configuration in the repository. More use of autonomic mechanisms and peer-to-peer dynamic reconfiguration significantly increases the need to support this type of information.

LCFG provides a mechanism for inserting values into configurations on the client, however, this process does not support the aspect composition operations available on the server, and hence any significant use of local information rapidly generates unresolvable conflicts in the overall configuration.

One requires a mechanism that supports late binding of those parameter values that are only available at runtime. The configuration environment should allow late-bound parameters to be declared and manipulated within the configuration description in the same way that static parameters are.

2.1.6 Contexts

It is often useful to be able to define several variations of a node configuration for use in different circumstances; for example, a laptop may need significantly different configurations, depending on whether it is connected to the local Ethernet, or connected via dial-up; a desktop machine may be used as a compute node at night, and may require a different configuration. Defining these variations as separate configurations, and then reconfiguring the node to move between these different contexts is not natural; indeed a laptop might need to make a context change while it is disconnected from the network and communication with a central configuration server is impossible.

LCFG provides a *context* mechanism which allows a number of different contexts to be defined. The client holds the configuration information for all contexts, and context switches can be made dynamically, even while disconnected from the network. This experimental implementation has proven extremely useful, but the semantics are very confusing, and as with late binding, the lack of full aspect composition on the client means that it is difficult to make extensive use of this feature without introducing conflicts. A similar mechanism which is more integrated with the configuration language is needed.

2.1.7 Peer-to-Peer

The LCFG deployment process is a simple matter of compiling configurations on the server and deploying them on the clients. There is no communication between the clients, and all configuration changes require a recompilation on the server. To support more autonomy (for scalability and fault-recovery), groups of clients need to be able to negotiate certain configuration changes among themselves, within the limits of a centrally defined policy. For example, if a server fails, a group of machines may decide autonomously to reconfigure one of their members as a replacement, without reference to the central configuration server. However, the central specification must still determine the policy; i.e. which nodes are eligible as the replacement.

2.2 Modelling Key Concepts in SmartFrog

SmartFrog is a framework to describe, configure and orchestrate distributed applications. It provides a declarative language that the runtime configuration engine can interpret and turn into complex, managed services. The latest, experimental iteration of the language introduces a number of concepts and constructs aimed at solving problems similar to the ones faced by the LCFG system.

The SmartFrog notation (see Appendix A for a more extensive description of the language) allows one to describe complex structures as the composition of simple elements, accompanied by an expression of the hierarchical relationships and dependencies between them. Like LCFG, it is a declarative, data description language, not a programming language. We believe that the SmartFrog language allows us to address most of the issues left open by previous languages, including LCFG itself, and offers a powerful, flexible language for clearly and concisely expressing complex configuration models.

2.2.1 Configuration descriptions

A fabric is woven out of many distinct yet often interdependent elements in the system, that are composed to form larger services. Basic aspects of the system can be defined as a collection of attribute-value pairs. These collections are referred to as descriptions. References can point to both simple values and descriptions. See, for example, the following excerpt.

```
// A template for a disk
Disk = {
  filesystem = ‘‘NTFS’’;
  size = 1024;
  partitionSizes = [10,5,5];
}
diskArray = {
   disk1 = Disk & { size = 2048; ... };
   disk2 = Disk & { size = disk1.size; ... };
   // disk2 has the same size as disk1
}
```

Variations on these basic aspects, such as a change in the value of an attribute, or the addition of extra content, are facilitated by the possibility to inherit from, or *augment*, existing templates with the desired values or constructs. This allows templates or descriptions to be progressively refined, by inheriting another instance. Both LCFG and SmartFrog are prototype-based languages [ABK+02]. An existing description may be augmented by changing the value of an attribute, or by adding extra content. This is illustrated below:

```
BigHPDisk = Disk & {
  size = 4096;
  make = ''HP'';
}

// would evaluate to :

BigHPDisk = {
  filesystem = ''NTFS'';
  partitionSizes = [10,5,5];
  size = 4096;
  make = ''HP'';
}
```

And :

```
extendedDiskArray = diskArray & {
  disk3 = {size = 4096; ...};
}

// would evaluate to :

extendedDiskArray = {
  disk1 = {size = 2048; ... };
  disk2 = {size = 2048; ... };
  disk3 = {size = 4096; ...};
}
```

Another example would be to define a web server node as a basic node augmented with the description of a web server application (see Section 2.3.1).

As will be seen below, the combination of augmentation with validation and set comprehension (see Section 2.2.3) facilitates aspect composition. This very flexible operation allows one to combine completely separate and distinct descriptions, and also provides the means to specify dependencies between these. For example, assuming you have a description of your system that models how DHCP allocates the IP addresses in the system, it is very easy to augment it with another description modelling the association of names to IP addresses (DNS) without having to modify the first one (as in Section 2.3.2).

2.2.2 Modifying and validating descriptions

The language also offers ways to modify the values in the descriptions, or collections of attribute-value pairs, with functions. These functions will typically escape to Java

for operations that are not part of the semantics of the language, such as simple string concatenation, or testing that an IP address specified in a description is valid.

Descriptions themselves can be validated to ensure correctness. By specifying predicates that can be used to validate the content of a description, we can ensure that a given collection of attributes maintains coherency according to some criteria. These predicates are preserved through the augmentation process, which means that we can define a template that will specify which valid values it should contain. When augmenting a description by modifying existing attributes, we still enforce that no illegal value can exist in the description. See the following example:

```
// large disks must have a size superior than 4096 MB
LargeDisk = Disk & {
  size = 8192;
} satisfying size > 4096;

// the following description is not valid:
LargeFATDisk = LargeDisk & {
  type = ''FAT'';
  size = 2048;
  // does not satisfy the inherited predicate size > 4096!
}
```

Furthermore, when combining two aspects in a single description we can define the condition each of them should satisfy for the combined result to be valid :

```
ContrastedPair = {
   bigDisk = { ... };
   smallDisk = { ... };
} satisfying bigDisk.size > smallDisk.size;
```

One could imagine a more complex example describing what a cluster is, by defining both the individual elements (what a valid working node is, what a valid server is, what a valid database is) but also the relationship between them, for example:

> A valid cluster is made of at least one defined server, one worker node, one database, and some job scheduling application deployed on every node.

Once this condition for the validity of a basic cluster has been defined, the original description can be further extended, e.g. a valid web-enabled cluster would require the existence of at least one webserver, yet would inherit all of the basic cluster's validity conditions.

2.2.3 Set comprehension: building list and data structures

Within the SmartFrog language, *set comprehension* allows one to define lists and collections of configuration descriptions according to a few simple constraints on the attribute values these descriptions contain. It provides a scalable and maintainable means for creating large collections of attributes.

If for example we want to build the list of all the small disks in the previous disk array[3]:

```
small = 2048;

// take every disk in the extendedDiskArray
// whose capacity is less than small
listOfSmallDisks = {
   diskName = disk;
} where (diskName = disk from extendedDiskArray)
        satisfying (disk.size <= small);

// would evaluate to:

listOfSmallDisks = {
  disk1 = {size = 2048; ... };
  disk2 = {size = 2048; ... };
} // disk3.size > small
```

In another context, we may also define a predicate that allows us to check whether the description of a node contains an application requiring a port to be opened in the firewall. It is then possible to collect all such nodes in a single structure, or to build the list of the ports to open.

Ordering on lists of attributes or on collections of descriptions can also be defined through functions.

Spanning maps as defined by LCFG may be reproduced in at least two ways in SmartFrog:

- either by providing a function which, applied to a description tree, would explore it and return a list of values for known attributes. For example, such a function could traverse the description of a system and return the MAC addresses of some nodes for the DHCP server;
- or by using set comprehension over known sets of value. There again, given the descriptions of the nodes in the system, it would be possible to iterate over such a set, and build the list of the MAC addresses of the nodes (see Section 2.1.3).

One main difference with spanning maps is that no explicit publication of the value is needed in either of the cases. This avoids having to change basic aspects when the collection of a new type of values is required by another part of the system. Therefore, to follow the previous example, the author of an application description need not care about publicising the fact that his application needs to punch a hole into the firewall; this remains the entire responsibility of the person defining the firewall.

[3] As will be seen below, set comprehension composes nicely with augmentation:

```
longerList =
listOfSmallDisks & {
  disk4 = {size = 8192;
...  };
  disk5 = {size = 1024;
...  };
}
```

would evaluate to :

```
longerList = {
  disk1 = {size = 2048;
...  };
  disk2 = {size = 2048;
...  };
  disk5 = {size = 1024;
...  };
}
```

Set comprehension can be further combined with references (e.g. by referencing the domain of the DHCP server we can build a list of MAC addresses of nodes belonging to the same domain, see Section 2.3.2), as shown below:

```
Nodes = {
  node1 = { domain = d1; macAddress = ... };
  node2 = { domain = d2; macAddress = ... };
  node3 = { domain = d2; macAddress = ... };
  node4 = { domain = d1; macAddress = ... };
}
// collect MAC addresses of nodes on domain d2
MACList = [node.macAddress] where node from Nodes
                            satisfying node.domain = d2;
```

Set comprehension also composes with augmentation. This second possibility puts a further emphasis on aspects separation: it is frequently the case that several parts of the system use the same aspects (typically nodes) to define different constructs. Both the descriptions of the DHCP serving of IP addresses and of a printing service would use the descriptions of nodes, potentially the same, yet they can independently use set comprehension to refine the descriptions of nodes they need to perform configuration actions on. A more extensive example can be found in the Section 2.3.3.

2.2.4 Using descriptions in the runtime configuration engine

With SmartFrog, some descriptions can be turned into running *configurator components* that have access to the same description framework. These configurator components typically read the set of static values they have been given on creation: a web server configurator component could simply read the port the web server application needs to listen to. Components may also communicate values between each other. References in the static description can therefore be used to point to dynamic values discovered or generated by other components. *Late bound* references (see Section 2.1.5) are not resolved as part of the static parsing of a description, but may at any time be dereferenced by the runtime component. As the runtime components have full access to the language descriptions from which they where created, they can also check the validity of the values they resolve dynamically. This check will often take the form of a call to the Java language in the configurator component, such as a type check on a dereferenced value. It can be more complex: for example, a component could be tailored to collect its IP address on startup and use a static value prescribed in the component description to perform a check (is the IP address in the right class A range, for example). Furthermore, it can contact another component to get a valid IP address if it fails these checks, or even add this value inside another component, which may in turn be notified of a new value and take appropriate actions.

Contexts (see Section 2.1.6) may work in the same manner; it is possible to define a collection of values describing different states of the system or of parts of it (a node being connected or disconnected from a given network for example), different deployment environments (an application can be deployed on a local host or on an entire cluster), etc.

These values can then be used by referencing them in affected descriptions where they will be meaningful.

A typical example is the hostname of a node: descriptions only need to refer to a given attribute hostname and this attribute itself points to a value given by the context. In many cases, context will be defined statically, but there again we can tie the discovery of the value of the attribute with the re-evaluation of the description.

When combining this approach with dynamic discovery protocols such as SLP, we can benefit from the solidity of both statically defined, valid models, and runtime, reactive values necessary for the sort of autonomous behaviour we are after.

2.2.5 Peer-to-peer

SmartFrog is a peer-to-peer system (see Section 2.1.7), and therefore helps address some of the centralization issues encountered by LCFG. Each peer in the SmartFrog system has the full capabilities of all the others: it can turn descriptions into configurator components, start or terminate them, etc. From that we may model our system using two diametrically opposed patterns:

- A main SmartFrog daemon may hold all the models and realize them on the fabric in an entirely centralized manner, thus “pushing” the descriptions on distributed daemons who act as passive deployers, in the LCFG manner.
- On the other hand, we can give to each peer complete access to all the models in the system, and in a way let them decide which configuration actions they should perform. This second approach requires distributed group membership protocols and rule engines which haven’t been explored as part of this study, but we believe such a framework could accomodate them easily. Take, for example, the following condition:

  > There should be two servers on this network segment.

  This would be realised by reaching an agreement between a set of peers as to which of them should hold a webserver, and the elected two would then take the description of the corresponding application and deploy it.

Most of the time we’ll be using an intermediate approach: templates will be distributed in a centralized manner, and the parameters evaluated at a more local level. For example, a central server would decide on the division of responsibilities between a subset of peers, and push the corresponding descriptions onto these peers, as well as allocating them a number of hosts to accomplish the given task or responsibility. More examples of such peer-to-peer patterns may be found in [AGP03].

2.3 The Three Primary Examples

2.3.1 OGSA model

Motivation

As stated above, the fundamental focus of the GridWeaver project is to identify and understand the challenges presented in managing the configuration of very large-scale infrastructures with high levels of heterogeneity and dynamism. Grid computing provides a pertinent and important scenario in which installations of this type are realised. Together, the Open Grid Service Architecture (OGSA) and the Globus Toolkit version 3 (GT3) represent key technologies for the future expansion and development of Grid-based computing environments.

Overview of OGSA and the Globus Toolkit

The incentive behind OGSA[4] is two-fold:

- To provide a consistent set of protocols and an environment to support the Grid computing infrastructure from earlier versions of Globus.
- To embrace and extend the web service standards which are being widely used in industry.

Existing web services do not address all of the needs of the Grid community. As such, OGSA can be viewed as an extension to the web service standards which have already been established. An OGSA-compliant service is a web service that satisfies additional requirements. The key additions are:

- Traditional web services are passive/stateless services (the implementation of state management in a web service is not covered by the core standards). OGSA introduces the explicit concept of service instances: each instance is uniquely named and can hold state.
- The Web Service Definition Language (WSDL) is central to the architecture. Currently, a web service is not required to provide a WSDL description of its interfaces. However, an OGSA-compliant service must do this.
- The naming mechanism for OGSA services has an indirection layer that is not present in web service standards. This allows individual Grid services to migrate between nodes over time in a transparent manner.

[4]There are two different acronyms used in this field OGSI (Open Grid Services Infrastructure) and OGSA (Open Grid Services Architecture). For the purposes of this discussion these will not be differentiated and the term OGSA will be used throughout. The concepts underlying OGSA are described in detail in [FKNT, TCF+].

- OGSA defines “Service Data”: a standard mechanism for OGSA services to publish information about their state (performance statistics, progress indicators, etc.).
- OGSA implements a security infrastructure derived from the security model available in earlier versions of Globus.

GT3 provides a hosting engine for OGSA-compliant services[5]. It is designed to run within a J2EE compliant web/application server, such as Tomcat [tom] or JBoss [jbo].

The Configuration Model

The configuration model which we have proposed for GT3 is derived from a number of different elements, representing well-defined tasks which must be completed during the instantiation and operation of the toolkit.

The elements which we have identified are:

(i) installation or deinstallation of the toolkit and applications on which it is dependent;

(ii) launch and termination of the toolkit;

(iii) derivation and specification of GT3 configuration parameters;

(iv) deployment and undeployment of OGSA services within the toolkit;

(v) configuration of deployed OGSA services within the toolkit;

(vi) configuration of GT3 security infrastructure;

(vii) configuration of standard OGSA services (for example, the Job Control Manager).

At the time of writing, item (v) has not been addressed in the OGSA specification and, for this reason, will not be considered in this report. Furthermore, it is the opinion of the authors that items (vi) and (vii) pose engineering rather than modelling difficulties. Thus, these items will not be addressed in this report.

From items (i)–(iv) of the list, we have derived a model of the configuration of GT3 based on the following configuration issues:

- Application packaging: This aspect models GT3 as a composition of specific, individual software components which have inter-dependencies between them. This aspect allows one to describe the installation and deinstallation of GT3.
- Services: This aspect models the general concept of a service.

[5] At the time of writing, a production release of GT3 was not available. Thus, the description presented in this report is based on the Beta release of the toolkit

- Hosting environments/servers: This element describes the pertinent parts of J2EE, required to administer Globus. For simplicity, we consider the specific example of GT3 operating within the Tomcat web server. However, we feel that the model is sufficiently general to allow other J2EE web servers to be covered with little additional effort.
- Toolkit configuration: This element describes the composition of Globus and the OGSA clients which it spawns.

Below, we consider each of these points in more detail.

Application packaging

A typical GT3 installation requires multiple software applications to be available. It is commonplace in a modern computing environment for applications to be bound into an installation package, for example Red Hat Package Manager (RPM) format [ABK+02]. In this report, we have restricted our attention to the RPM system. However, as observed in [ABK+02], RPM has many similarities with other packaging tools and these could be modelled in an analogous manner.

In our simple model, an RPM will be identified by two attributes that prescribe the name and the version of the package. In the RPM system, this is sufficient to uniquely identify a package (although on multiple platforms, one would also need to attribute the build architecture to the package).

In SF2, the RPM template is:

```
RPMPackage = {
  name:String;
  version:String;
  requires [] satisfying all x::RPMPackage;
  conflicts [] satisfying all x::RPMPackage;
}
  satifying name.defined and version.defined;
```

Typically, a package will have dependencies on other packages, from which it derives common functionality (for example, on Linux systems `curses` provides terminal functionality which is used by the majority of terminal applications). In our model, a package will be bound to a list of other packages on which it depends.

It will also be bound to a list of packages with which it is known to conflict. This allows us to express incompatibilities between specific versions of applications. For example, a Java Runtime Environment may not work on a specific version of the Linux kernel:

```
java-1.4.1v3-1 = RPMPackage & {
  name = "java";
  version = "1.4.1";
  conflicts = ["kernel-2.2.*"];
}
```

In the majority of cases, it is desirable to validate that these dependencies are satisfied at the compile stage: that is, the collection of packages prescribed for installation is self-consistent. Validation of these constraints is the responsibility of the SmartFrog compiler.

On occassion, it may not be possible to determine the dependencies which a package has (for example, if the package description is not available to an author of a description).

Dependencies and conflicts may not be explicitly prescribed in the model and it may not be possible for the compiler to dereference these values until the configuration description is deployed. In this case, the dependencies must be late-bound at the client node, only being validated at that point. As noted in Section 2.1.5, such a circumstance is likely to generate unresolved conflicts. These must be tackled dynamically by the configuration system at runtime.

The complete model of the GT3 installation is presented in Appendix B.

Services

The term service has a level of ambiguity which must be eliminated before we can specify the modelling elements. The simplest type of services (that we consider) is a daemon process which runs on a single node. We call such a service, a “local service”. While the concept of a local service is simple, particular instances may represent extensive applications such as web servers and database servers – the fundamental restriction is that the service runs on a single node.

In our model, local services are the building blocks from which we construct more complex services. Complex services may be composed of multiple other services and be distributed across many nodes within the fabric. As such, a service has attributes describing its name, version, and a list of other services on which it depends. Represented in SmartFrog 2, the service prototype is:

```
Service = {
  name:String;
  dependencies [] satisfying all x::Service;
}
  satifying name.defined;
```

A local service is an extension of a service that has additional attributes describing which node the service is running on and the RPM from which the service is installed. From the service prototype, a local service is derived:

```
LocalService = Service & {
    host::Node;
    installedBy [] satisfying all x::RPMPackage;
}
```

A complete SF2 description of the service model is presented in Appendix B.

Host environments/servers

For this example, we have restricted our model of the hosting environment to those elements which are necessary in order to describe the configuration of Globus to be deployed within J2EE.

In the model, a J2EE-compliant Web application is derived from a generic service with additional attributes that prescribe the minimum heap needed by the application and lists describing the supported MIME types and required JAR archives:

```
J2EEWebApplication = Service & {
  heapSize:Integer;
  mimeTypes [] satisfying all x::MIMEType;
  classPathJars [] satisfying all x::JarFile;
}
  satisfying heapSize.defined => [maxHeapSize, 0].gt;
```

Simple consistency checks are implemented to ensure that attributes, such as `heapSize`, are assigned valid entries.

Having defined a Web application, it is sufficient that we represent a Web server simply as a collection of web applications:

```
J2EEWebServer = {
    webApplications []
        satisfying all x::J2EEWebApplication;
}
```

A full description of the Tomcat Web Server is presented in Appendix B.

Toolkit configuration

The Globus toolkit is an example of a J2EE-compliant Web application that hosts OGSA-compliant services. We collect the list of services associated with a single instance of the toolkit into a container which is hosted on a single node. This may be represented in SF2 as:

```
OGSAContainer = LocalService & {
  deployedServices []
    satisfying all x::OGSAService;
}
```

It is then a simple matter to compose the various elements which make up the Globus toolkit:

```
GlobusToolkit = J2EEWebApplication & OGSAContainer & {
    name = "Globus Toolkit Version 3";
    installedBy = GlobusToolkitRPM;
    mimeTypes = [MIMEType & {extension="wsdl";
```

```
                              type="text/xml"},
                MIMEType & {extension="xsd";
                              type="text/xml"}];
    maxHeapSize = 128;
    classPathJars = []; /* List of JAR files required by
                         * GT3 to be specified
                         */
}
```

Additional comments on the model

One will note that a description of the underlying network topology and hardware specification is absent from this model. It is important that the modelling language allows this since, in a large infrastructure, the responsibility for configuring a service such as GT3 may be delegated to an individual who does not have either the jurisdiction or expertise to affect the network configuration. Furthermore, as noted in Section 2.1.4, over-specification of a configuration increases the likelihood of conflicts occurring and inhibits the flexibility of autonomic systems to re-configure in the event of a component failure. We present a model of the cluster infrastructure in the next example, Section 2.3.2.

The model above is derived in a service-centric manner: services are treated as a key element. Based on this model, a description of a specific cluster would fundamentally be a list of the services that are to be available. From this and other aspects (such as cluster configuration and the security model) information pertaining to the configuration of individual nodes would be derived implicitly using language constructs.

An alternative approach would be to focus on the node entity and attribute each instance of a node description with the processes or services that should be installed on it. A simple example of these two different approaches is presented below:

```
// Service-centric approach to modeling a cluster.

SimpleService = LocalService & {
  node = NodeA;
  /* other configuration details ... */
};

AnotherSimpleService = Service & {
  node = NodeB;
 /* other configuration details ... */
};



// Node-centric approach to modelling a cluster.

NodeA = Node & {
  processes = {SimpleService};
  /* other configuration details ... */
};

NodeB = Node & {
```

```
  processes = {AnotherSimpleService};
  /* other configuration details ... */
};
```

Both these approaches are equally valid. They are alternative strategies for presenting essentially the same configuration information. Which approach is better depends upon the type of system being modelled. The SF2 set comprehension functionality is effectively a query language that allows either one of these views to be transformed into the other.

Our model also illustrates how different types of dependency information can be expressed. These constraints are expressed as part of the model. However, the responsibility for checking the validity of such constraints remains the responsibility of the compiler. Approaches to the treatment and implementation of contingencies to be considered in the event of constraint validation failure have not been addressed in this example.

2.3.2 Naming service model

Motivation

The ability to quickly and reliably configure the network infrastructure of a fabric is fundamental in almost every practical computing environment. In all but the smallest installations, it is a technical and potentially time consuming task with cross-nodal dependencies to be resolved and complex services to be deployed. In this example, we have defined a model that allows one to describe a Transmission Control Protocol/Internet Protocol (TCP/IP) network and to configure a naming service on it.

The example illustrates mechanisms which are relevant to a number of the challenges described in Sections 2.1 and 2.2, including the following:

- devolution of responsibility for different aspects of configuration;
- constraint specification and validation;
- pan-nodal configuration (for example, the LCFG spanning map construction).

Overview of the naming service and IP infrastructure

In this model, a node is attributed with an IP address from a DHCP mapping held in a DHCP table, and this IP address is bound to a hostname by a DNS mapping.

The model which we present is a simplification of a typical TCP/IP network installation. The simplifications which have been made represent subtleties in the engineering of a practical network and do not detract from the pertinence of the design. Key simplifications are as follows:

- limit our description to a single subnet;
- ignore hostname aliases in the naming service;

- do not represent configuration parameters for the DNS and DHCP services;
- define a single DNS server for the domain;
- assume that a network interface can be uniquely identified by its Media Access Control (MAC) address.

When devising the description, two key points were apparent:

(i) no user entry of information should be duplicated;

(ii) the model should be amenable to a delegation of responsibility between multiple administrators.

To implement (i), we demonstrate how attributes can be accessed from different parts of the description using a reference construction in the modelling language.

To illustrate (ii), we present two versions of the model:

- in the first version, all information pertaining to the network is the responsibility of a single administrator;
- in the second version, configuration information relating to the hardware assets is the responsibility of one administrator, while allocation of hostnames to nodes (via the naming service) is the responsibility of another.

Components of the network

A subnet is a collection of contiguous IP addresses. These IP addresses can be deduced from the subnet's base IP address and its netmask. Our subnet description represents this information, and also has a placeholder for a single router, via which nodes within the subnet may communicate with the extended network. This can be represented in SF2 as follows:

```
Subnet = {
  baseIPAddress:String;
  netMask:Integer;

  routerIP:String;
}
```

The nodes which are assigned to the subnet are, from the viewpoint of the network, a collection of network interfaces. Each interface has a MAC address and a placeholder for a hostname (which is populated from DNS using a spanning map-type construction (see Section 2.1.3)):

```
Interface = {
  macAddress::MACAddress;
  hostname::Hostname;
}
}
Node = {} satisfying all x::Interface;
```

The DNS naming service is a collection of maps binding hostnames to IP addresses. These maps are held in a lookup table, attributed to an internet domain (note that we have simplified the situation by having a single DNS table per internet domain):

```
DNSMapping = {
  hostname::Hostname;
  ipAddress::String;
}

DNSTable = {
  domainName:String;
  mappings = {} satisfying all x::DNSMapping;
}
```

In a similar fashion to the naming service, DHCP is a collection of maps binding an IP address to a MAC address. These maps are collected into a table which is attributed to a particular subnet:

```
DHCPMapping = {
  ipAddress:String;
  macAddress::MACAddress;
}

DHCPTable = {
  subnet::Subnet;
  mappings = {} satisfying x::DHCPMapping;
}
```

Using these descriptions as building blocks, we have constructed a model of a generic TCP/IP network configuration. As the network configuration is composed from multiple descriptions with potentially different authors, it is sensible to implement consistency checking at this point. To demonstrate this, we have provided two simple validation policies for the description. Expressed in SF2, the template network configuration is:

```
IPSystem = {
  domains = {} satisfying all x:String;
  subnets = {} satisfying all x::Subnet;
  nodes = {} satisfying all x::Node;
  dhcpTables = {} satisfying all x::DHCPTable;
  dnsTables = {} satisfying all x::DNSTable;
}
satisfying
(
```

```
  // Constraint 1: There is a DHCP table for each subnet
  // in the system
  for all subnet in subnets,
    [exists] dchpTable in dhcpTables where
       (dhcpTable.subnet.baseIPAddress ==
        subnet.baseIPAddress),

  // Constraint 2: Every interface is registered in a
  // DHCP table, and the IP address of the interface is
  // valid within its designated subnet.
  for all node in nodes,
    for all interface in node,
      [exists] subnet in subnets where
        (dhcptable from dhcpTables,
         mapping from dhcptable.mappings,
         mapping.macAddress == interface.macAddress,
         [dhcptable.mappings.mapping.ipaddress,
          subnet.baseIPAddress].startsWith)

  // ... other constraints go here.
)
```

The two instances of constraint satisfaction constitute bounded searches over a dynamically constructed table, looking for instances of satisfaction of the prescribed boolean expression. The construction has a strong analogue with a query specification in a database application.

Having established the component descriptions that form the basis of the model, we have illustrated how these may be realised through two different examples.

In the first example, we have constructed a model in which all information is explicitly stored in the node description. To construct the descriptions of DHCP and DNS, reference is made to the attributes defined in the node description. This ensures that no duplication of information occurs. This is a typical scenario with no delegation of ownership for different elements of the configuration description.

In the second example, the configuration of a node is composed from attributes held in two different descriptions. Static information about a node's network interface is held in a node description, while the prescription of an IP address to the node is defined in the naming service description. Again, there is no duplication of information – the DHCP configuration is specified implicitly with actual attribute values being dereferenced from the node and naming service descriptions. This second example demonstrates how the model can represent the delegation of responsibility for different elements of configuration, that typically occurs within a large organisation.

Version 1: Single point of administration

This example details an organisation of information which is most amenable to management of the network infrastructure by a single individual. Below, we define a node description which allows one to record the MAC address plus allocate a hostname and IP address in a single step. Since the basic definition of an interface does not provide a

placeholder for an IP address, we have firstly enhanced the basic component set for the model, adding a new prototype:

```
ExtendedInterface = Interface & {
  ipAddress:String;
}
```

With this additional prototype, the node pool looks like:

```
// Owned by a single administrator
allNodes = {
  ...,
  beethoven = Node & {eth0 = allInterfaces.beethoven0;},
  mozart = Node & {eth0 = allInterfaces.mozart0;},
  ...
}

allInterfaces = {
  ...
  beethoven0 = ExtendedInterface & {
    macAddress = "00-10-B5-0A-9E-BF";
    hostname = gridweaverHost & {
      hostName = "beethoven";
    }
    ipAddress = "15.144.27.190";
  },

  mozart0 = ExtendedInterface & {
    macAddress = "07-56-C4-0F-1F-AA";
    hostname = gridweaverHost & {
      hostName = "mozart";
    };
    ipAddress = "15.144.27.192";
  },
  ...
}
```

All the information required to create the DNS and DHCP table is held in the node (and interface) descriptions. Thus, the DNS and DHCP tables can both be constructed implicitly by referencing the appropriate content from the set of interfaces. For example, to construct the DNS table for **gridweaver.org**, the SF2 `where` construction is used to extract and collate corresponding IP address and hostname entries for every interface contained within the domain description:

```
dnsTables = {
  ...
  gridweaverDNSTable = DNSTable & {
    domain = allDomains.gridweaverDomain;
    mappings = {
      -- = DNSMapping & {
        hostname = interface.hostname;
        ipAddress = interface.ipAddress;
```

```
    } where (
      node from allNodes,
      interface from node,
      interface.hostname.domain == domain
    )
  ...
}
```

A similar construction is used to create the DHCP table. In this case, a search is performed over the set of all interfaces in the subnet and all mappings in DNS, identifying matches between hostname entries in the two sets. For each match found, a map is created in the DHCP table taking the MAC address from the interface and the IP address from the corresponding DNS mapping:

```
allDHCPTables = {
  jcmbDHCPTable = DHCPTable & {
    subnetName = jcmbSubnet;
    mappings = {
      -- = DHCPMapping & {
        ipAddress = dnsmapping.ipAddress.;
        macAddress = interface.macAddress;
      } where (
        node in allNodes,
        interface in node,
        dnstable in dnsTables,
        dnsmappings in dnstable.mappings,
        dnsmapping in dnsmappings,
        dnsmapping.hostname.hostName ==
          interface.hostname.hostName,
        [dnsmapping.hostname.ipAddress,
         subnet.baseIPAddress, subnet.netMask].matches
      );
    }
  }
}
```

Version 2: Devolved responsibility for hardware and DNS

In this second example, we organise configuration data into two separate descriptions, representing the domain of control of two different administrators. This separation is necessary since, in the SmartFrog language, a description is the finest resolution at which ownership can be specified. In this example, one person owns the description of a node and the other person owns the description of the naming service. Considered separately, either of these descriptions is incomplete – only when the two descriptions are combined do we obtain a complete specification of the configuration.

When devising a model that is amenable to a devolved responsibility or ownership, one must be especially careful to establish the provenance of key modelling attributes. Firstly, we have considered the hardware administrator who is responsible for hardware assets (for example, workstation, printers, and so on) on the network. One of the tasks performed by this person is to add new nodes to the network. Consequently, within our model this

person has been given responsibility for describing nodes (and their interfaces). They are able to ascertain the MAC address of each new interface and allocate a hostname to it[6]. In SF2, an excerpt from the description of the node pool is as follows:

```
// Owned by the hardware assets administrator
allNodes = {
  ...,
  beethoven = Node & {eth0 = allInterfaces.beethoven0;},
  mozart = Node & {eth0 = allInterfaces.mozart0;},
  ...
}

allInterfaces = {
  ...,
  beethoven0 = Interface & {
    macAddress = "00-10-B5-0A-9E-BF";
    hostname = gridweaverHost & {
      hostName = "beethoven";
    };
  },

  mozart0 = Interface & {
    macAddress = "07-56-C4-0F-1F-AA";
    hostname = gridweaverHost & {
      hostName = "mozart";
    };
  },
  ...
}

gridweaverHost = Hostname & {
  domain = "gridweaver.org";
}
```

Having added new nodes to the pool description, the hardware administrator advises the naming service administrator who attributes each new node (that is, interface) with an appropriate IP address (within DNS):

```
// Owned by the naming service administrator
dnsTables = {
  gridweaverDNSTable = DNSTable & {
    domain = domains.gridweaverDomain;
    mappings = {
      ...
      -- =  DNSMapping & {
        hostname = allNodes.beethoven.eth0.hostname;
        ipAddress = "15.144.27.190";
      }

      -- = DNSMapping & {
        hostname = allNodes.mozart.eth0.hostname;
```

[6] In practice, this person would make many more such choices. For example:- location, owner, operating system, software to install, and so on.

```
        ipAddress = "15.144.27.192";
      }
      ...
    }
  }
}
```

To complete the definition, the DHCP table needs to be created. Since all of the relevant information is available, no further manual additions need to be made to the configuration – the correct mapping can be deduced from the information already provided by the hardware administrator and naming service administrator. This is a similar construction to that presented in the previous example:

```
dhcpTables = {
  jcmbDHCPTable = DHCPTable & {
    subnetName = jcmbSubnet;
    mappings = {
      -- = DHCPMapping & {
        ipAddress = dnsmapping.ipaddress.;
        macAddress = interface.macAddress;
      } where (
        interface in node,
        node in allNodes,
        dnstable in dnsTables,
        dnsmappings in dnstable.mappings,
        dnsmapping in dnsmappings,
        dnsmapping.hostname.hostName ==
          interface.hostname.hostName,
        [dnsmapping.hostname.ipAddress,
         subnet.baseIPAddress, subnet.netMask].matches
      );
    }
  }
}
```

Additional comments on the model

Both versions of the naming service description require per-node attributes (for example, the MAC address) to be collated across a set of node descriptions. This functionality has strong similarities to the LCFG Spanning Map construction. However, one key difference between the LCFG implementation and the example presented here is that the source (in this case individual node descriptions) do not explicitly publish attributes. The collation operation is performed entirely by the DNS and DHCP descriptions that subscribes to the information, the source (node description) is unaffected.

The complete SF2 model for the naming service can be found in Appendix B.

2.3.3 Printing model

Motivation

Configuration of large computing infrastructures implies the configuration of services that may affect the entire fabric, or large portions of it. These services are typically difficult to represent in conventional configuration management systems for several reasons.

Firstly it is often impossible for the service, and even less for the configuration system, to make assumptions on the number and state of the clients of the service. The state of the clients may be modified for a number of reasons:

- client nodes may simply fail and thus not require service anymore;
- they may subscribe to or unsubscribe from a service at any time;
- fabric-wide policies may change and thus cause a client to need a service, or to be denied one;
- a client may request a new or upgraded version of the service because its own internal state has changed (e.g. new libraries installed, new version of the operating system . . .).

The second difficulty is due to the complexity of the dependencies between subsystems involved to ensure the proper behaviour of a fabric-wide service:

- the service will often require a set of dedicated nodes to be available, ready to act as clients;
- the service itself may depend on the existence of another service in the system to run;
- clients of a service themselves may very quickly build dependencies on a set of layered services.

A further difficulty arises with the presence, in the same fabric, of aspects of configuration that are often treated or seen as independent, but have indeed side-effects on one another. One would like these aspects to compose nicely, and at the very least to be able to quickly modify an aspect of the configuration in a coherent manner. The archetypal example is that of the implementation of security policies in a fabric; services such as authentication may have to be put in place. This affects a very large number of nodes in the fabric.

Printing Service : a candidate for complex modelling

A printing service has a few properties of interest to us:

- It manipulates abstract entities: print queues are the abstraction given to users of the printing system, encapsulating the relationship between a print server (to which print jobs are submitted) and an actual printer.

- It affects three different types of basic configuration elements:
  - printers, as hardware elements with a set of capabilities;
  - nodes, either as print servers or as client machine connecting to the service;
  - users, which are given a set of permissions to access the service.

  Each of those can come in and out of existence in the system at any point in time: for example print servers fail or are being upgraded, printers jam or run out of toner.
- Complex dependencies exist between the printing service and some key low-level aspects of the whole configuration of the fabric. Typically a naming service of some sort is required for print servers and printers, as well as a directory or a database to hold user information.
- The service also has some inner dependencies; a queue requires both the existence of a printserver and the printers it is associated to. Somehow we want the model of our system to reflect such shared fate between elements. This requires that we accommodate two conflicting tensions:
  - On the one hand we need to be explicit and refer to concrete elements, as the existence of a queue implies actual configuration actions triggered on a print server whose name must be known, such as the printing daemon being started.
  - On the other hand, we'd like to abstract the link between a queue and a print server or a printer. It allows us to design a system able to take corrective actions so that a queue is preserved when, for example, a print server fails and is replaced by a different node.
- A correctly configured printing service relies on a few assumptions that we would like to be able to model, and specify as constraints for the system. Typically one would want to specify the fact that there should be at least always one print server in the system, or, for reliability purposes, one print server and at least one spare, etc. Such constraints may also depend on other aspects of the system: we might want to specify in our model that the number of print queues served by print servers does not exceed the storage capacity of these servers, so that the corresponding spool directories can always be created.
- Finally, transversal concerns such as new security restrictions or permissions may impact the whole printing service, and require the reconfiguration of either clients or print servers themselves. New types of users may appear in the system that we'd like to accommodate, and so on.

Overview of a real printing system

Our modelling example is partly based on the printing system at the University of Edinburgh, School of Informatics (part of the DICE system). The printing system in place accommodates different sites and their own subnets; different types of users with different printing permissions; and heterogeneous systems due to legacy printing applications

or security mechanisms present and used on the site. Below, we have established the context for our example and documented the assumptions that underline and constrain our model.

Networking is set up so that each site or building has its own private subnet, to which are connected the printers, and one print server serving them. Print servers have a dual interface and are also connected to the University network. Print servers act as a DHCP server for the printers they serve.

Queues are associated to a single printer. All information for the queue management comes from an LDAP directory, which is used to provide information for both clients and servers: translating from a queue name to an actual print server on the client side, and from a queue name to a printer and its capabilities on the print server itself. The LDAP server also contains information about the users of the printing service and their permissions on the system.

The software used in the printing system is LPRng. Mainly this printing daemon needs to be able to:

- get information about queues;
- get the printing permissions of users;
- get information about the printers themselves: for example this information will be used to process user options for a print job according to the actual capabilities of the printer (duplex, color, and so on).

The model needs to reflect the link between these source of data.

Modelling a printing service

Our modelling exercise attempts to capture such a complex system in a coherent manner, and in a single model. For the purpose of this exercise and to focus on the modelling aspect, we're not taking into account legacy applications, or some singularities of the LDAP directory used in the system. In this section, we present excerpts of the model, highlighting important issues. The full model is presented in Appendix B.

We first define the basic elements of our system. We build upon the models developed in Section 2.3.2. Both a Printer and a Print Server are extensions of a Node description with network configuration specified using the naming service model.

A queue can easily be represented as a name, and a link to a printer. A list of all queues intended to be provided to users in the system can be defined.

```
Queue = {
  name;
  printer::Printer;
}
Queues = {} satisfying all x [x,Queue].extends;
```

A printer is a node connected to one of the printing subnets, from which it is allocated an IP address. It is augmented by a description that defines its printing capabilities and typical user options. A list of all printers in the system can be also defined.

```
Printer = Node & {
  printingSubnet::Subnet;
  ipAddress;
  ppdFile;
  Zaliases = {} satisfying all x:String;
}
Printers = {} satisfying all x::Printer
```

The print server itself can be described as a standard node, with a second network interface living on a printing subnet. It also has a DHCP table to serve IP addresses for the printers connected on this subnet. Finally, it serves the queues associated to the printers on this subnet. We can also define a list of all print servers on our system, and enforce the existence of at least one print server per subnet.

```
PrintServer = Node & {
  eth1::Interface;
  printingSubnet = jcmb_PrintingSubnet;
  dhcpTable = DHCPTable & {
    subnet = 'printingSubnet;
    // collect IP and mac of all printers on this subnet
    dhcpMappings = {
      -- = {
        ipAddress = p.ipAddress;
        macAddress = p.eth0.macAddress;
      } where
          (p from Printers,
           p.printingSubnet == subnet);
    };
  };
  // the queues served are all the queues serving
  // a printer on the same subnet
  queuesServed = {}
    where (q from Queues,
           q.printer.printingSubnet == subnet);
  ...
}
// The list of print servers, to be
// populated by the actual values
PrintServers = {} satisfying
          for all s in PrintingSubnets,
          [exists] printserver,
                  (printserver.printingSubnet == s);
```

In a way we are layering two naming services: when joining the printing network printers, we get their IP address through the print server, which got its own IP address from a regular DHCP server on the University network. As can be seen, the print server model makes no assumptions on the number of printers it serves, nor does it hardwire the names

of the queues served. Our modelling allows us to specify simply that the print server serves any printer on its subnet.

It is also easy to show that, from this set of simple models, we can recompose information needed by other parts of the system. There is obviously a discrepancy between the model and the components realising the system, but the manipulation of models allows us to bridge the gap. For example, if we were to model the content of the LDAP directory that runs on the system, we could define the structure below:

```
LDAPDirectory = {
  // queues to print server mappings
  queuesToPrintserver = {
    -- = {
      queueName = queue.name;
      servingPrintserver = printserver;
    }
  } where
      queue from queues,
      printserver from PrintServers,
      [printserver.queuesServed,queue].contains;
  // queues to printer mappings
  queuesToPrinter = ‘Queues;
```

The first construct in the LDAP Directory model gathers all mappings from a queue name to the printserver that serves it, and the mappings from a queue name to its printer (a relationship already enclosed in the Queue model).

It is also extremely easy to layer new aspects on top of such a model. We can, for example, define users as having a set of queues they can submit jobs to.

```
User = {
  name;
  queuesAllowed = {}::Queue;
}
Users = {}::User;
```

With the printing software used at the University, the permissions are set by allowing users to individual printers. There again it is simple to model such mappings between users and printers:

```
Permissions = {
  -- = {
    username = user.name;
    printersAllowed = [p] where
                        (q from user.queuesAllowed,
                         p = q.printer);
    }
} where user from Users;
```

This model presents the following advantages:

- we’ve easily defined and reused the basic elements;
- we were able to layer abstractions : for example we defined the structure of a correctly configured print server without having to make, in our model, explicit references to the queues that will be served;
- we’ve achieved some degree of separation of concern: our lists of elements (such as Printers, Users, etc.) may be maintained independently by distinct person or entities.

2.4 Conclusions from the Modelling Exercise

The models we presented in the previous sections are an example aimed at showing some of the concepts we think are essential to configuration modelling, and the capabilities offered by the SmartFrog language. We believe the language proposed not only can replace LCFG as a configuration description language, but offers more power and flexibility:

- *Basic aspects*: It is easy to define basic elements and compose them in more complex structures. Furthermore, once a number of templates have been defined, it is easy to reduce the number of lists of these basic elements to maintain.
- *Composition of basic aspects*: The augmentation operator gives us the ability to build upon these models in a way that facilitates devolved management. Aggregate models may be defined in a very flexible manner, and may reflect high-level abstractions, without explicitly tying them to low-level constructs.
- *Lists and Spanning maps*: With set comprehension, we can very easily define collections of elements according to predicates over data structures. In addition, we can augment these collections with required constructs without modifying the elements they are pointing to, further enhancing our capabilities to separate responsibilities and devolve management of some aspects of the model. More complex operations on data structures can also always be performed through the use of functions.
- *Validation*: We provide constructs in the language to define what valid configurations are, and how aggregate structures should be composed. They act as both predicates of validity and structured documentation.
- *Context*: Through parameterization of required attributes and the possibility to easily reference to other data structures, we can represent context in the static descriptions; such information may later be used by the runtime system.
- *Peer to peer*: From such models as defined in the previous sections, it is very easy to derive data structures to be used in components realising the system, as can be shown in [BKMP03], and implement and use such things as *late binding* of references and *context* (through reference to high level elements in the description, and runtime components interpreting these attributes and being notified of the changes in their values).

Although we believe the SmartFrog language is a significant leap forward in the domain of configuration description languages, a number of things are still open to further debate and investigation:

- The language is still in an experimental state, in that it has been used by a very limited number of users; we still need to determine how usable, flexible and simple some of the constructs are, and how “natural” the language may be to users.
- We would like to represent temporal aspects of attributes and their values. It would be convenient to define the relationship between values or even data structures with time, such as

The numbers of ports open in the intranet should be minimal during the night.

- Constraints satisfaction would also help us define a system with a few high level statements. Attempts at maintaining high level conditions such as those presented in 2.1.4 have been made using the SmartFrog runtime framework; however the language does not provide any support for this at the moment.
- Modelling remains a thorny problem; one of the drawbacks of the SmartFrog language's flexibility is that it gives the user many different ways to describe the same thing (see 2.3.1 for example). We are still investigating patterns for the modelling; we also believe the language will be an ideal vector for experimentations on modelling and will help progress in that area.

As part of this investigation on the modelling we present in the following section an integrated architecture for our two configuration deployment technologies, LCFG and SmartFrog. As mentioned before, the language itself has no notion of the meaning of the attributes it manipulates. The intended goal is therefore that such an environment be able to understand the model, and realize it into a running system. Our final report also presents a model deployed on this architecture [BKMP03].

Chapter 3

The Combined LCFG/SmartFrog Framework

In this section we provide a detailed description of the architecture for the integration of SmartFrog and LCFG to create a testbed environment.

3.1 Overview

The two systems SmartFrog and LCFG represent complementary solutions convering different aspects of the task of controlling the configuration of a fabric[1]. This complementary nature provokes an idea of an integrated environment in which we exploit the benefits from both packages. Specifically, we identify the following key benefits of the two systems:

- SmartFrog encourages pan-nodal, peer-to-peer interoperability;
- SmartFrog has a mature, versatile, declarative language;
- LCFG has a comprehensive resource of components which have been used to manage the configuration of a number of real fabrics;
- SmartFrog supports a flexible set of component life-cycles;
- LCFG has spawned a wealth of information regarding practical aspects of the implementation of a configuration management infrastructure.

Based on the intent to reuse and exploit these features of the two packages, we have designed an integrated configuration environment, presented in Figure 3.1. Three categories of elements exist within Figure 3.1: configuration descriptions, management systems (MS), and components.

The configuration descriptions express the intended target configuration of the fabric, either in part or as a whole. Each description is purely declarative and may either be written in the SmartFrog 1 (SF1) language or in the LCFG language.

[1] Detailed descriptions of both these systems can be found in [ABK+02].

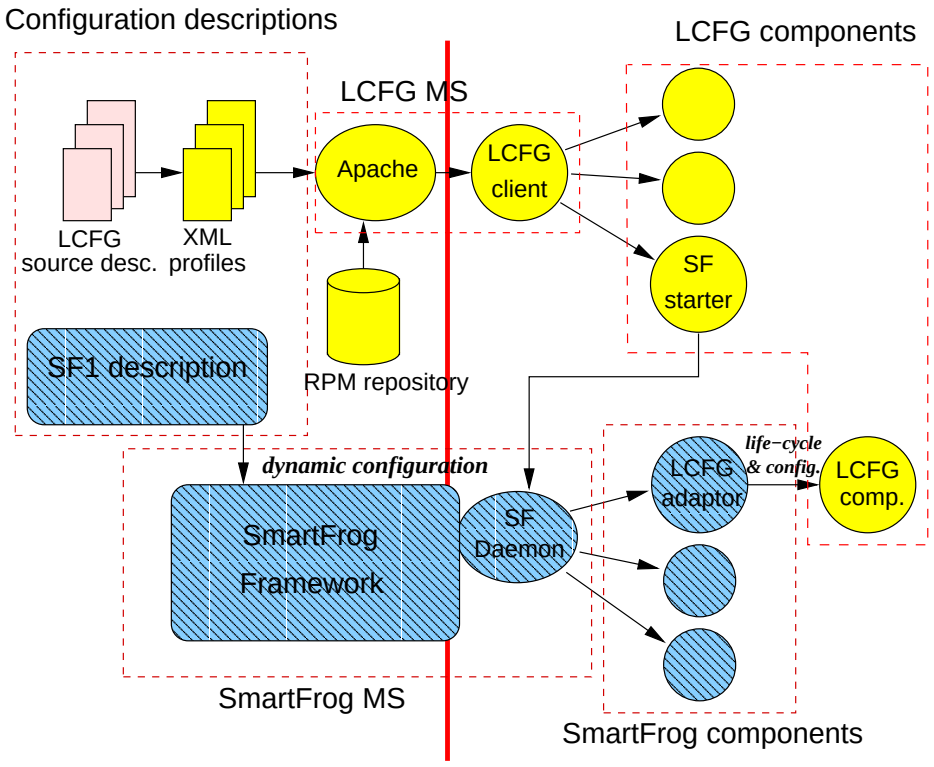

Figure 3.1: Diagrammatic representation of the architecture for the integration of LCFG and SmartFrog.

Two management systems exist: these are LCFG and SmartFrog. They each take a configuration description, evaluate it, and generate appropriate instructions which are then passed to the configuration components.

The configuration components make necessary changes to system files, registries, etc., in order to affect the configuration expressed in the descriptions.

Items within Figure 3.1 are shaded to indicate their origin, as follows:

- elements that have a hashed background relate to SmartFrog;
- elements that have a uniform shade background relate to LCFG.

A solid line partitions the elements of the figure into two sets. Those elements on the left of the line are implemented on the server side, while elements on the right of the red line are implemented on the client side.

Components within the figure are also implicitly partitioned into two phases relating to what we have named “static configuration” and “dynamic configuration”.

Static configuration A phrase used when making reference to configuration information which is implemented during the initialisation of the fabric and which, once deployed, does not alter.

Static configuration information is represented by the upper branch of the diagram: that is *LCFG Source descriptions, XML Profiles, . . . , SF starter*.

Within our architecture, static configuration is principally handled by the LCFG system. An installation package which allows the rapid deployment of the LCFG system onto a new cluster has been developed as part of the project. Further details of this package can be found in Section 3.4.

Dynamic configuration A phrase used in reference to aspects of configuration which may change during the normal operation of the infrastructure. This includes both anticipated changes, and changes in response to a designated combination of events or circumstance. The SmartFrog framework is utilised to manage all instances of dynamic configuration within the testbed.

Dynamic configuration is represented by the lower branch within the diagram: that is *SF1 description, SmartFrog Framework, ..., LCFG comp*.

The dynamic configuration of the installation is implemented in a case-by-case manner. An example of dynamic configuration is provided by the GPrint application [BKMP03].

In our testbed, before dynamic configuration aspects can be implemented, the SmartFrog framework must be in place. The installation of the SmartFrog framework is performed by LCFG during the static configuration phase. An LCFG component named *SF starter* (discussed in Section 3.2.7) is responsible for initialising the SmartFrog daemon on each node within the fabric (as denoted by the arrow leading from *SF starter* to *SF Daemon* on the client side of the diagram). This implies a temporal dependency between static configuration and dynamic configuration. In the testbed, we enforce this by prescribing that all static configuration be completed before any dynamic configuration aspects commence.

The architecture presented in Figure 3.1 is one particular representation of the testbed. In Section 3.3, we describe several additional components which would be desirable to implement in a future revision so as to further improve the usability of the infrastructure. However, these additions have not been implemented at this time.

3.2 Components of the architecture

In this section, we look in greater detail at the individual elements represented in Figure 3.1.

3.2.1 LCFG source descriptions (*server-side*)

Availability:	Developed as part of LCFG Installation Package.
Format:	Text files – LCFG description languguage.
Interfaces with:	Created/modified by a text editor.
	Passed to the LCFG server

The static configuration of the testbed is described in a collection of LCFG source descriptions called profiles. A profile exists for each node within the fabric, though common configuration information may be shared between a number of profiles by collecting the information into a separate header file and then including it using the CPP `#include` directive.

Source descriptions are written using the LCFG configuration language.

3.2.2 XML profiles (*server-side*)

Availability:	Available – created by LCFG server.
Format:	XML files.
Interfaces with:	Created/modified by LCFG server. Passed to LCFG components.

LCFG source profiles are compiled into per node XML files by the LCFG profile server. These are published on a web server from where they are read by LCFG client daemons. These XML files are created automatically by LCFG and no modifications to this process have been made for the testbed.

3.2.3 RPM repository (*server-side*)

Availability:	Constructed as part of LCFG Installation Package.
Format:	Redhat Package Manager data files .
Interfaces with:	Read by LCFG *updaterpms* and *rpmcache* components

The method by which LCFG distributes application files (for example: executable, template LCFG configuration files, and the Linux kernel) is the Red Hat Packaging environment, commonly referred to as RPM. Many RPMs are provided as part of the typical Red Hat Linux distribution, though additional packages specific to this testbed have been created (for example, the SmartFrog installation package).

3.2.4 Apache (*server-side*)

Availability:	Third-party Open Source software.
Format:	Daemon application.
Interfaces with:	Hosting node contains LCFG profiles and a Red Hat RPM repository. Profiles are read by the LCFG *client* component. RPMs are read by the LCFG *updaterpms* and *rpmcache*

components.

Compiled XML profiles and Red Hat Packages (RPMs) from the LCFG Installation Package (see Section 3.4 are placed on an Apache web server, ready to be pulled across by an LCFG client.

The Apache web service is installed and configured on the LCFG server as part of the static configuration process: note that an LCFG component called *apache* which can configure Apache exists as part of the normal LCFG distribution.

3.2.5 LCFG client (*client side*)

Availability:	Available as part of LCFG distribution.
Format:	Daemon application.
Interfaces with:	Pulls down LCFG profiles via HTTP. Notifies LCFG components that their XML profile has changed.

An LCFG client runs on each node within the testbed. It is responsible for automatically reading the particular node's XML profile from the LCFG server node each time that it is changed. The client converts the XML profile into a DBM file and then notifies any components whose configuration has been affected by the update process.

As LCFG is used to configure aspects of static configuration on each node, an LCFG client daemon will run on every compute node within the fabric. This client is installed and deployed as part of the LCFG Installation Process described in Section 3.4.

The application exists as part of the normal LCFG distribution and no modification is required for this demonstrator.

3.2.6 LCFG components (*client side*)

Availability:	Core set available as part of LCFG. Additional components will be created as part of GridWeaver project.
Format:	Bourne/Perl shell script applications.
Interfaces with:	Queries DBM format of the LCFG profile for relevant configuration information. Implements configuration changes in a component-specific manner.

As a practical environment which is at the core of the School of Informatics configuration strategy, LCFG contains a powerful resource for implementing actual configuration changes via *LCFG components*. As noted above, in this testbed, LCFG components are managed either by the LCFG system or by SmartFrog, though never by both systems.

Existing LCFG components have been utilised in the demonstrator wherever possible. However, several additional components have been developed: these are described elsewhere in this document.

3.2.7 SF Starter (*client side*)

Availability:	Developed for the testbed.
Output form:	LCFG component written using Bourne shell script.
Interfaces with:	Deploys the SmartFrog daemon and configures its operation according to information p

This is an LCFG component which can be used to deploy and manage the SF Daemon. It will provide:

- The ability to start and stop the SF Daemon.
- The ability to set three configuration variables that control the daemon.
  - `classpath` – the classpath the daemon uses.
  - `javahome` – the location of the JRE to use.
  - `home` – the root location of the SmartFrog installation.

3.2.8 SF1 description (*server side*)

Availability:	Written as part of a specific application of the testbed.
Output form:	Text file – SmartFrog 1 description language
Interfaces with:	Created in a text editor (GNU Emacs editing mode provided). Interpreted by the SmartFrog Framework.

Dynamic configuration of the testbed is composed as a description written in the SmartFrog 1 (SF1) language. The great majority of the information contained within this file is specific to the intended application. An example is provided by GPrint [BKMP03].

3.2.9 The SmartFrog framework (*server and client side*)

Availability:	Available as part of the SmartFrog distribution.
Format:	Java container environment.
Interfaces with:	Controls SmartFrog components. Implements the configuration specification from the SF1 description.

For aspects of configuration which either span multiple nodes or represent dynamic configuration, SmartFrog is the more suitable environment in which to operate. In contrast to LCFG, which manages configuration in a node-centric manner, SmartFrog allows relationships between nodes to be expressed in a very clean and natural manner.

In this testbed, each node runs a copy of the SmartFrog daemon. This is initialised, by an LCFG component *SF starter*, during the static configuration phase.

3.2.10 SmartFrog components

Availability:	A core set is available as part of SmartFrog. Additional components have been created for the testbed.
Format:	Java applications.
Interfaces with:	Controlled by the SmartFrog framework. Implements configuration changes in a component-specific manner.

Analogous to LCFG, SmartFrog implements configuration change using component applications running on the target nodes. A number of components are provided as part of the SmartFrog distribution. Additional components have been developed for the testbed and example applications deployed on it: these are described elsewhere in this document and in [BKMP03].

3.2.11 LCFG adaptor

Availability:	Created as part of the testbed.
Format:	Java applications.
Interfaces with:	Controlled by the SmartFrog framework. Alternative to LCFG client as an interface to LCFG components.

A SmartFrog component, called the *LCFG adaptor*, is used to control LCFG components which are to be managed by SmartFrog, instead of the LCFG system.

This component facilitates more dynamic interaction with LCFG components, exploiting the diverse life-cycle options available within SmartFrog, plus allowing SmartFrog to apply dynamic modifications to LCFG component configuration data.

The adaptor:

- allows context values in LCFG to be changed;
- provides an interface between the SmartFrog daemon and an LCFG component, by exposing the usual LCFG component methods (for example, `start` and `stop`);
- implements an algorithm for converting an SF1 description of configuration data into the equivalent LCFG XML profile. This profile is then passed to the LCFG client in order to initiate reconfiguration of the particular component.

3.3 Possible additions to the architecture

It is the longer-term aim of the GridWeaver team that the description of the configuration of a fabric be encapsulated into a single model, developed using the SmartFrog 2 language

(SF2) or a derivative thereof. This description would incorporate both LCFG source descriptions and the current SF1 description. However at present, in the absence of an SF2 compiler (or a translator from SF2 to SF1), this implementation is not possible.

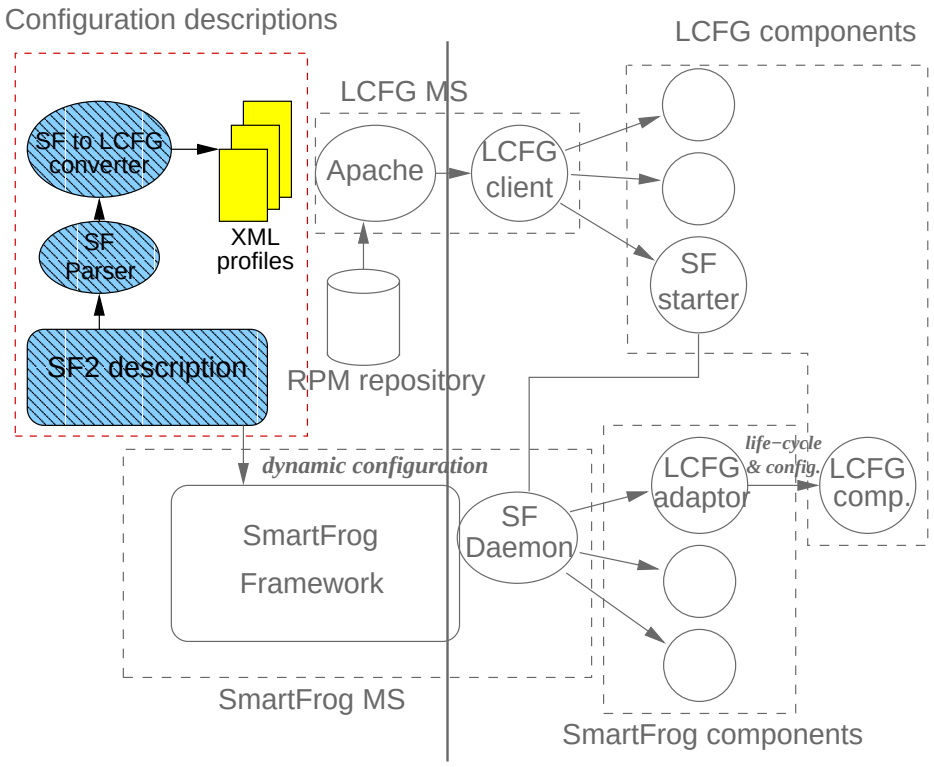

Figure 3.2: Diagrammatic representation of a possible architecture which encapsulates the description of the configuration into a model developed using the SmartFrog 2 language.

A modified architecture which incorporates our suggested changes is presented in Figure 3.2. This figure is very similar to Figure 3.1 – the only differences occurring within the "Configuration description" container.

Within the "Configuration description" container, the SF1 description and LCFG source files have been replaced by an SF2 description. Furthermore, two new elements called the "SF Parser" and the "SF → LCFG converter" are included.

The role of the SF Parser is to filter the canonical (expanded) form of the SF2 configuration description into two streams: (i) information which is intended for the LCFG system; (ii) information which is intended for the SmartFrog framework.

Subsequent to extraction from the SF2 description, information which is intended to be processed by the LCFG system is passed to the SF → LCFG converter. This module transcribes the attribute assignments into an XML formatted datafile which is suitable for the LCFG client daemon.

3.4 LCFG installation package

3.4.1 Introduction

As part of the creation of a testbed environment, it was necessary to package the LCFG system so that it could rapidly and simply be deployed onto a new cluster. Since installation of client nodes becomes straightforward once an LCFG server is available on a cluster, this task effectively amounted to devising a procedure by which a self-configuring LCFG server could be installed from scratch.

The package which has been developed allows either an LCFG server or client to be installed and configured on a cluster in about 15 minutes (for a typical current desktop PC specification). Two pre-requisites to the installation exist: (i) a web server must be accessible from the subnet (this will host RPM packages and an initial compiled XML profile description for the server), and (ii) a DHCP server must be accessible from the subnet (this will allow nodes to receive network information, pertaining to the configuration of the cluster and the location of LCFG services, at boot time).

3.4.2 Installation procedure for server and client nodes

With the LCFG Installation Package, the following simple steps are required to build an LCFG server on a cluster:

1. On an existing LCFG server, create a source description for the new server and compile this to obtain an XML node profile. Publish this profile on a web server which is accessible from the target node.
2. Power up candidate node and boot from the LCFG Installation CD-ROM.
3. Select installation method “cdrom with dhcp”.
4. Enter the location (URL) of the XML profile description (created in Step 1). Installation will then proceed automatically.
5. Once the new LCFG server is running, copy the server description onto it and set the node to run in server mode (configuring itself).
6. Reboot the node.

At this point, one has an autonomous LCFG server. One may then proceed to install individual client nodes on the network. For each client node, perform the following steps:

1. Create a source description of the node on the local LCFG server.
2. Power up the candidate node and boot from the LCFG Installation CD-ROM.
3. Select installation method “cdrom with dhcp”. Installation will proceed automatically.

3.4.3 Components of the package

The installation package is composed from a number of distinct component elements. These are:

- A CD-ROM containing a bootable LCFG/Linux system. The CD-ROM contains a copy of the Linux kernel (version 2.4) plus essential basic components required by LCFG during the installation procedure.

  The following components were identified as being essential to the installation process: syslog; dns; inet; client; amd; homedir; symlink; updaterpms; kernel; auth; sshd; cron; hardware; init; ntp; pam; network; lilo; fstab; logserver; apache; and server. For a description of these components, refer to [And01].

- A CD-ROM containing a base set of RPMs required for LCFG and Linux. This repository must be published onto a web server prior to commencing the installation of a new cluster.

3.4.4 LCFG source profiles and header files

To facilitate the creation of node configuration descriptions, a set of portable LCFG source profiles and header files has been provided. Header files provided are:

- `common.h`: Site-specific attribute declarations;
- `server.h`: LCFG server-specific configuration attributes;
- `client.h`: LCFG client-specific configuration attributes;
- `redhat71.h`: Operating system-specific entries (in this case for Red Hat Linux, version 7.1);
- `hp_epc_42.h`: Platform-specific entries (in this case, for the HP EPC platform);
- `smartfrog.h`: Attributes that control the configuration of the SmartFrog deployment;
- `globus.h`: Attributes that control the configuration of the Globus Toolkit Version 3.

By collecting information pertaining to different aspects of the configuration of the fabric into separate header files, we greatly simplify the process of creating new node descriptions. For example, this is an example of a typical source profile for a client on our testbed:

```
/* LCFG Source profile for client1.hpl.lcfg.org */
#include <mutate.h>
#include <gw_client.h>
```

```
/* Site-specific information */
#include <locations/gw_hpl.h>

/* Add node-specific hardware requirements here */
#include <hardware/gw_hp_epc_42.h>

/* Services */
#include <gw_smartfrog.h>
#include <gw_globus.h>

/* Add node-specific resources here. */
```

As one can see, this client has been configured to run SmartFrog and Globus Toolkit 3 simply by including the appropriate module. This example is considered in more detail in [BKMP03].

3.4.5 Further information

More information about the specifics of the LCFG Installation Package including, for example, the content of the base RPM repository or the source code to the LCFG header files can be obtained by contacting the GridWeaver project team [Edi].

Appendix A

The SF2 language

This appendix is a presentation of the SF2 language. We quickly present some of the concepts motivating the creation of the language, then we detail the main constructs of the language. Complete examples exercising the language may be found in Appendix B.

A.1 A brief rationale for the design

The core assumption behind SmartFrog (and LCFG) is that defining data (as attributes) is enough to describe system configuration: configuration states, service lifecyles and dependencies between various service components. In that respect, SmartFrog is not a programming language, in the same way as XML or RSL.

The semantics of the language itself need consider only the definition of attribute sets, how they are formed, and how they compose. As we focus on building data, we also ignore the interpretation of the data by subsequent tools. Typically, the language can be turned into a normalised form which can be easily interpreted and used by tools such as a configuration engine; however the semantics of the language are only concerned with manipulating data.

To operate on the values defined in the language, we provide controlled escape mechanisms to Java, which also give us the possibility to extend the notation with new functions and base predicates. However, this remains outside the scope of the core language semantics.

A.2 Language Aspects

A.2.1 Attributes

A configuration is defined as a set of attributes, each of which is uniquely named inside a description. An attribute may have a value, such as:

- a value of some simple type: integer, string, float;
- a collection (list, set, ...) of values.

In the following example, a disk is described as the collection of a filesystem “NTFS”, a size in integer, and a vector representing the partition sizes[1]:

```
disk = {
  filesystem = "NTFS";
  size = 20;
  partitionSizes = [10,5,5];
}
```

The two following sections present other possible types of values: these are collections and references.

A.2.2 Composition

To build anything more than a basic description, we need to construct compositions of attribute sets. To do this, attributes may have values which are themselves attribute sets, thus allowing one to write nested descriptions.

We can, for example, define a collection of disks:

```
diskArray = {
  disk1 = {
    filesystem = "NTFS";
    size = 72;
    partitionSizes = [36,36];
  };
  disk2 = {
    filesystem = "FAT32";
    size = 36;
    partitionSizes = [36];
  };
  // etc.
  disk3 = { ...
  };
};
```

A.2.3 Dependencies

It is extremely frequent either to reuse the same value in several places, or to make one attribute’s value dependant on other values. In general we need to be able to express relationships between values. To allow this, we define links between attribute values. We use naming paths which may be relative or absolute. Examples of both types of reference are given in the following:

[1] It is interesting to note here that the language itself has no notion of what the size means, neither does it know the units used. It is up to the tools interpreting the language to understand the meaning of these attributes.

```
maximumSize = 72;
diskArray = {
  disk1 = {
    size = ROOT.maximumSize; // an absolute path to 72
    ...
  };
  disk2 = {
    size = 36;
    ...
  };
  disk3 = {
    size = disk2.size; // a relative path to 36
    ...
  };

};
// we can traverse the tree structure:
disk3size = diskArray.disk3.size; // a relative path to 36.
```

A.2.4 Composition: set comprehension

When building tree structures and hierarchies of objects, we often have to create vast collections (such as all nodes on a network, all disks on a file server, etc.). Set comprehension in SmartFrog provides scalable and maintainable means for creating such large collections.

In the following simple example, we create an array of identical disks from a list of names:

```
diskNames = ["disk1", "disk2", "disk3", ... ];
diskArray = {
  disk = {
    size = 20;
    filesystem = "NTFS";
  } | disk from diskNames
}
// and we can still use links:
disk3size = diskArray.disk3.size;  // points to 20
```

In this case the diskArray construct would evaluate to the following description:

```
diskArray = {
  disk1 = {
    size = 20;
    filesystem = "NTFS";
  }
  disk2 = {
```

```
    size = 20;
    filesystem = "NTFS";
  }
  disk3 = {
    size = 20;
    filesystem = "NTFS";
  }
}
```

A.2.5 Modification – the augment operator

Configuration descriptions constantly need to be modified. The author of an initial configuration description will tend to include a set of default values for the majority of the attributes. However, subsequent users of the description will have to change the value of parameters, to tailor the initial description for customisation, licensing purposes, to adapt it to a different environment (for example, specific class of IP addresses, different locale, and so on) or to suit user preferences. For example, it is possible to write a generic description of a web server although to create a description of the Apache web server, one will need to enhance this description. A system administrator might take a template which they download from the Apache website and modify it according to the specific requirements of a local intranet and make it compatible with the rest of the environment. Then, an end-user might take the web server template offered by the administrator and modify it so as to serve a given directory on their desktop machine. This example goes to show that any attribute in a description may change, arbitrarily and without structure, according to the requirements of a sequence of users of a description. However, what we do not want to do is have to edit, at each level, the same configuration file – offering the same level of information regardless of the user. We need an operator that allows us to modify the configuration (rather than edit the description) and that reflects both incremental changes and different hierarchies of user. SmartFrog provides the *augment* operator (represented by the symbol &) which offers a flexible way to modify configuration, through incremental extension, inheritance and parameterisation. In the following, we present the main capabilities of the operator and show how it composes with the previously defined language constructs.

Adding attributes

To start simply, values may be overwritten, and new attributes may be added to an existing description:

```
Disk = {
  filesystem = "NTFS";
  size = 20;
}
// A bigger disk is a disk augmented with a size 40
BiggerDisk = Disk & {
```

```
  size = 40;
}

// An HP is a  bigger disk is augmented with an
// attribute "make"
HPDisk = BiggerDisk & {
  make = "HP";
}
```

The last two constructs would evaluate to:

```
BiggerDisk = {
  filesystem = "NTFS";
  size = 40;
}

HPDisk = {
  filesystem = "NTFS";
  size = 40;
  make = "HP";
}
```

The values of attributes can also be descriptions, so one can augment a collection of attributes with another description. For example we can augment our previous disk array with another disk:

```
extendedDiskArray = diskArray & {
    disk4 = { size = 18; ...};
}
```

This extended disk array would evaluate to:

```
extendedDiskArray = {
  disk1 = { size = 72; ...} ;
  disk2 = { size = 36; ...} ;
  disk3 = { size = 36; ...} ;
  disk4 = { size = 18; ...} ;
}
```

Interaction with links

Links and augmentation interact in an interesting way. Typically, changing a value implies that any dependencies also change.

```
disks = {
  disk1 = { size = 40; ...};
  disk2 = { size = 80; ...};
  disk3 = { size = disk1.size + 20; ... };
  // disk3 has a size 60 = 40 + 20
}
differentDisks = disks & {
  disk1 = {size = 70};
  // this time disk3 has a size 90 = 70 + 20
}
```

Interaction with functions

Frequently, over-writing basic attributes is not correct: when the construct is augmented we would like to augment a list, augment a sum, etc. SmartFrog allows the use of arbitrary functions on attribute values, functions which can also be written by users of the language. The default behaviour is to apply right projection of the augmented attributes' values.

In the following example the disk has a required size, which is augmented by the application installed on the disk. In short, augmenting a disk with an application adds the disk capacity requirements of the application to the running total.

```
disk = { requiredSize: Integer; ... }

// we define a generic application requirement
applicationRequirement = {
  requiredSize =.+ 0;
  // on augmentation, add the right hand value to 0
  ...
}
// and we specialise this requirement through augmentation:
oracleRequirement = applicationRequirement & {
  requiredSize = 40;
}
apacheRequirement = applicationRequirement & {
  requiredSize = 10;
}

myDisk = disk & oracleRequirement & apacheRequirement;
// myDisk.requiredSize = 50; (0 + 40 + 10);
```

It is to be noted that & is not commutative when it comes to such composition (precisely because of this "right projection"), therefore we need to consider carefully how to combine and propagate modifiers during augmentation. The generic rule is:

```
For basic values:

{a =.f 24; ...} & {a =.g 57; ...}
                ==>
       {a =.f[24,57].g;  ...}

General case:
{a = X; b = Y; } & {a = X'; c = Z;}
                 ==>
   {a = (X & X'); b = Y; c = Z;}
```

A.2.6 Validation

We also provide a way to ensure that augmentations are valid (in the same manner as one might want to check for types). Schemas (and indeed types) have two roles:

- as predicates of validity: is an attribute value of the right type, is an attribute in the right range?
- as structured documentation, indicating expected values to users, whether a value is optional or required, and providing tools such as GUI-based editors with a list of possible attributes[2].

We can attach a predicate to a collection of attribute values, introducing it with the keyword *satisfying*.

In the following, a disk is defined whose capacity must be greater than a given minimum:

```
disk = {
   size = 40;
   minSize = 40;
} satisfying size >= minSize;
```

Composition with augmentation

Predicates have to be inherited through augmentation, as values themselves are. In this way, augmentation of two constructs maintains all vaidation policies from the original constructs.

```
miniDisk = disk & {
  size = 10;
}    // invalid: size = 10 < minSize = 20
     // the language compiler will propagate an error
```

[2] Again, this does not provide the semantic interpretation of these attributes.

```
anotherDisk = disk & {
  minSize = 20;
}   // valid: size = 40 >= minSize = 20

yetAnotherDisk = miniDisk & {
   minSize = 5
};  // valid: size = miniDisk.size = 10 >= minSize = 5
```

The general semantics of the composition of augmentation and the *satisfying* predicate are:

```
(X satisfying P) & (Y satisfying P)
                ==>
   (X & Y) satisfying (P AND P)
```

Predicates – syntactic shortcuts

Abbreviated notation may be substituted for some frequently used predicate constructions:

- Predicates of type: for example, attribute “disk” is an Integer. The normal expression:

  ```
  disk = {size=0, ...} satisfying size.Integer;
  ```

  is equivalent to

  ```
  disk = {size:Integer = 0; ...}
  ```

- Inheritance predicates: if description “B” is constructed from description “A” augmented by additional data, then the predicate:

  ```
  {B = ... } satisfying [B,A].extends
  ```

  is equivalent to

  ```
  {B::A = ... }
  ```

Predicates: Quantification

It is possible to define predicates with quantification over attributes. For example, if one defines a set of resources and wants to define the predicate “*All values inside the resource description are positive integers*”, this can be expressed in SmartFrog, as follows:

```
resources = {
  servers = 34;
  firewalls = 45 ;
  ...;
} satisfying all x. (x.Integer and x > 0)
```

Appendix B

Example SF2 models

B.1 OGSA model

In this section, we present the SF2 source code that complements the OGSA model described in Section 2.3.1. At the end of the section, we include two examples of instantiation of the deployment of the Globus Toolkit in a service-centric manner. These examples pertain to a stand-alone service and a service hosted within a container environment, respectively.

```
/**
 * Example 1: OGSA model
 *
 * Configuration of a Grid infrastructure based on
 * OGSA and GT3.
 */


// The following definitions represent a subset of
// the networking model required for this example.

// A node is simply a collection of interfaces
Interface = {
  macAddress::MACAddress;     // unique physical
                              // identifier for the
                              // interface
  primaryHostname::Hostname;
}

Node = {} satisfying all x::Interface;

Port = {
    portNumber:Integer
}
satisfying
(
  portNumber.Integer and portNumber > 0;
)
```

```
Network = {} satisfying all x::Nodes;


/*************************************************
 * Application packaging
 *************************************************/
RPMPackage = {
  name:String;
  version:String;
  requires [] satisfying all x [x, RPMPackage].extends;
  conflicts [] satisfying all x [x, RPMPackage].extends;
}
satifying name.defined and version.defined;


// The following definitions represent the specific
//  packages we are interested in installing.

JREPackage = RPMPackage & {
  name = "Java Runtime Environment";
  version = "1.4.1_01";
}

TomcatPackage = RPMPackage & {
  name = "Tomcat";
  version = "4.0.4";
  requires = [JREPackage];
}

GlobusToolkitPackage = RPMPackage & {
  name = "Globus Toolkit";
  version = "3.0 alpha";
  requires = [JREPackage];
}


/*************************************************
 * Services
 *************************************************/
Service = {
  name:String;
  dependencies [] satisfying all x::Service;
}
satifying name.defined;

LocalService = Service & {
    host::Node;
    installedBy [] satisfying all x::RPMPackage;
    autostart:Boolean;
}


/*************************************************
 * Hosting environment/servers
 *************************************************/
```

```
MIMEType = {
  extension:String;
  type:String;
}

JarFile = {
  name:String;
  location:String;
}

J2EEWebApplication = Service & {
  heapSize:Integer;
  mimeTypes [] satisfying all x::MIMEType;
  classPathJars [] satisfying all x::JarFile;
}
  satisfying heapSize.defined => [maxHeapSize, 0].gt;


J2EEWebServer = {
  port::Port;
  webApplications []
    satisfying all x::J2EEWebApplication;
}


/*************************************************
 * Toolkit configuration
 ************************************************/
Tomcat = J2EEWebServer & LocalService & {
  port = Port & { portNumber = 8080};
  installedBy = [TomcatPackage];
}

OGSAService = Service & {}

OGSAContainer = LocalService & {
  deployedServices []
    satisfying all x::OGSAService;
}

GlobusToolkit = J2EEWebApplication & OGSAContainer & {
  name = "Globus Toolkit Version 3";
  installedBy = GlobusToolkitRPM;
  mimeTypes = [MIMEType & {extension="wsdl";
                           type="text/xml"},
               MIMEType & {extension="xsd";
                           type="text/xml"}];
  maxHeapSize = 128;
  classPathJars = [JarFile & {name="cog-jglobus.jar",
                              location=""},
                   JarFile & {name="puretls.jar",
                              location=""},
                   JarFile & {name="cryptix32.jar",
                              location=""},
                   JarFile & {name="cryptix-asn1.jar",
```

```
                                location=""},
                  // Other JAR files go here ...
                 ];
}


/*************************************************
 * Examples of instantiating GT3
 *************************************************/
//
// To deploy Globus Toolkit as standalone application
//
GlobusToolkitStandalone = GlobusToolkit & {
  host = "beethoven";
  autoStart = True;
}

//
// Alternative, to deploy Globus Toolkit within a
// Tomcat container.
//
MyTomcat = Tomcat & {
  host = "elgar";
  autoStart = True;
  webApplications = [GlobusToolkit];
}
```

B.2 Naming service model

In this section, we present the SF2 source code that complements the naming service model described in Section 2.3.2. The source code is divided into three fragments:

- Section B.2.1 contains the description of the basic modelling components which constitute the network configuration;
- Section B.2.2 contains a description best suited to configuration from a single source (that is, by a single administrator);
- Section B.2.3 contains a configuration designed for an environment in which two different administrators operate with different domains of responsibility for: the hardware assets of the fabric and the configuration of the naming service, respectively.

B.2.1 Modelling components

```
/**
 * Example 2: Network and naming service model
 *
 * Simple configuration for a TCP/IP network with
 * naming service.
 */


// A subnet is a collection of contiguous IP addresses
Subnet = {
  baseIPAddress:String;
  netMask:Integer;

  routerIP:String;            // only one router per
      // subnet is allowed in
      // this example
}


// A node is simply a collection of interfaces
Interface = {
  macAddress::MACAddress;     // unique physical
      // identifier for the
      // interface
  primaryHostname::Hostname;
}

Node = {} satisfying all x::Interface;


// hostname consists of domain and local element
Hostname = {
  name:String;
  domain:String;              // the domain for this
```

```
      // host
}


// name/IP address mappings collated into a table held
// on DNS server
DNSServer = Node;

DNSMapping = {
  hostname::Hostname;
  ipAddress:String;
}

DNSTable = {
  domain:String;
  mappings = {} satisfying all x::DNSMapping;
}


// MAC/IP address mappings collated into a DHCP
// table. In this example, must have precisely one DHCP
// table per subnet.
DHCPMapping = {
  macAddress:String;
  ipAddress:String;
}

DHCPTable = {
  subnet::Subnet;
  mappings = {} satisfying all x::DHCPMapping;
}


// The building blocks described above are used to
// construct simple prototype of IP network
// configuration
IPSystem = {
  domains = {} satisfying all x:String;
  subnets = {} satisfying all x::Subnet;
  nodes = {} satisfying all x::Node;
  dhcpTables = {} satisfying all x::DHCPTable;
  dnsTables = {} satisfying all x::DNSTable;
}
satisfying
(
  // Constraint 1: There is a DHCP table for each subnet
  // in the system
  for all subnet in subnets,
    [exists] dchpTable in dhcpTables where
       (dhcpTable.subnet.baseIPAddress ==
        subnet.baseIPAddress),

  // Constraint 2: Every interface is registered in a
  // DHCP table, and the IP address of the interface is
  // valid within its designated subnet.
```

```
  for all node in nodes,
    for all interface in node,
      [exists] subnet in subnets where
        (dhcptable from dhcpTables,
         mapping from dhcptable.mappings,
         mapping.macAddress == interface.macAddress,
         [dhcptable.mappings.mapping.ipaddress,
          subnet.baseIPAddress].startsWith)

  // Constraint 3: Every interface domain is valid.
  for all node in nodes,
    for all interface in node,
      [exists] domain in domains where
        (interface.primaryHostname.domain == domain);

  // Constraint 4: All IP addresses are in the correct
  // subnet
  for all dhcpTable in dhcpTables, mappings in
          dhcpTables, subnet in subnets,
    (subnet.baseIPAddress ==
     dhcpTable.subnet.baseIPAddress)  =>
      [mappings.ipaddress,
       subnet.baseIPAddress].startsWith;

  // Constraint 5: All MAC address are unique.
  for all node in nodes,
    interfaceName = interface in node,
      for all otherNode in nodes,
        differentInterfaceName = differentInterface
          in otherNode,
        (otherNode.differentInterfaceName !=
         node.interfaceName) => (interface.macAddress !=
         differentInterface.macAddress);

  // ... other constraints go here.
)
```

B.2.2 Single point of administration

```
/*************************************************
 * Version 1: Single point of administration
 *************************************************/

// Allow IP address to be stored with the interface.
ExtendedInterface = Interface & {
  ipAddress:String;
}

allNodes = {
  beethoven = Node & {eth0 = allInterfaces.beethoven0;},
  mozart = Node & {eth0 = allInterfaces.mozart0;},
  elgar = DNSServer & {eth0 = allInterfaces.elgar0;}
}
```

```
allInterfaces = {
  beethoven0 = ExtendedInterface & {
    macAddress = "00-10-B5-0A-9E-BF";
    hostname = gridweaverHost & {
      hostName = "beethoven";
    }
    ipAddress = "15.144.27.190";
  },

  mozart0 = ExtendedInterface & {
    macAddress = "07-56-C4-0F-1F-AA";
    hostname = gridweaverHost & {
      hostName = "mozart";
    };
    ipAddress = "15.144.27.192";
  },

  elgar0 = ExtendedInterface & {
    macAddress = "07-56-C4-2F-BE-11";
    hostname = gridweaverHost & {
      hostName = "elgar";
    };
    ipAddress = "15.144.27.193";
  }
}


// Construct DNS table and DHCP table based on the
// information in allNodes
dnsTables = {
  gridweaverDNSTable = DNSTable & {
    domain = allDomains.gridweaverDomain;
    mappings = {
      -- = DNSMapping & {
        hostname = interface.hostname;
        ipAddress = interface.ipAddress;
    } where (
      node from nodes,
      interface from node,
      interface.hostname.domain == domain
    )
}

allDHCPTables = {
  jcmbDHCPTable = DHCPTable & {
    subnetName = jcmbSubnet;
    mappings = {
      -- = DHCPMapping & {
        ipAddress = dnsmapping.ipAddress.;
        macAddress = interface.macAddress;
      } where (
        interface in node,
        node in nodes,
        dnstable in dnsTables,
        dnsmappings in dnstable.mappings,
```

```
        dnsmapping in dnsmappings,
        dnsmapping.hostname.hostName ==
          interface.hostname.hostName,
        [dnsmapping.hostname.ipAddress,
         subnet.baseIPAddress, subnet.netMask].matches
      );
    }
  }
}

// Build the network
gridweaverIPSystem = IPSystem & {
  domains = allDomains;
  subnets = allSubnets;
  nodes = allNodes;
  dnsTables = allDNSTables;
  dhcpTables = allDHCPTables;
}

// ... and we have our configuration.
```

B.2.3 Devolved responsibility for hardware and DNS

```
/**************************************************
 * Version 2: Devolved responsibility for hardware and
 * naming service.
 **************************************************/

// Owned by the hardware assets administrator
gridweaverHost = Hostname & {
  domain = "gridweaver.org";
}

allNodes = {
  beethoven = Node & {eth0 = allInterfaces.beethoven0;},
  mozart = Node & {eth0 = allInterfaces.mozart0;},
  elgar = DNSServer & {eth0 = allInterfaces.elgar0;}
}

allInterfaces = {
  beethoven0 = Interface & {
    macAddress = "00-10-B5-0A-9E-BF";
    hostname = gridweaverHost & {
      hostName = "beethoven";
    };
  },

  mozart0 = Interface & {
    macAddress = "07-56-C4-0F-1F-AA";
    hostname = gridweaverHost & {
      hostName = "mozart";
    };
  },
```

```
  elgar0 = Interface & {
    macAddress = "07-56-C4-2F-BE-11";
    hostname = gridweaverHost & {
      hostName = "elgar";
    };
  }
}

allSubnets  = {
  royalAlbertHall = Subnet & {
    baseIPAddress = "15.144.27.0";
    netMask = 8;
    defaultRouterIPAddress = "15.144.24.1";
  }
}

allDomains = {
  gridweaver = gridweaverHost.domain
}


// Owned by the naming service administrator
dnsTables = {
  gridweaverDNSTable = DNSTable & {
    domain = allDomains.gridweaver;
    mappings = {
      -- =  DNSMapping & {
        hostname = allNodes.beethoven.eth0.hostname;
        ipAddress = "15.144.27.190";
      },

      -- = DNSMapping & {
        hostname = allNodes.mozart.eth0.hostname;
        ipAddress = "15.144.27.192";
      },

      -- = DNSMapping & {
        hostname = allNodes.elgar.eth0.hostname;
        ipAddress = "15.144.27.193";
      },
    }
  }
}


// DHCP table is collated from the information which is
// already available.
dhcpTables = {
  rahDHCPTable = DHCPTable & {
    subnetName = royalAlbertHall;
    mappings = {
      -- = DHCPMapping & {
        ipAddress = dnsmapping.ipaddress.;
        macAddress = interface.macAddress;
      } where (
```

```
        node in allNodes,
        interface in node,
        dnstable in dnsTables,
        dnsmappings in dnstable.mappings,
        dnsmapping in dnsmappings,
        dnsmapping.hostname.hostName ==
          interface.hostname.hostName,
        [dnsmapping.hostname.ipAddress,
         subnet.baseIPAddress, subnet.netMask].matches
      );
    }
  }
}

// Build the network
gridweaverIPSystem = IPSystem & {
  domains = allDomains;
  subnets = allSubnets;
  nodes = allNodes;
  dnsTables = allDNSTables;
  dhcpTables = allDHCPTables;
}

// ... and we have our configuration.
```

B.3 Printing service model

In this section, we present the SF2 source code describing the complete printing service model described in Section 2.3.3. The source code is relatively short, but builds in reality on the naming model defined in the previous section of this appendix. In that respect, we inherit the definition of all the basic components (see B.2.1) and all other constructs. For simplicity, we've omitted part of the descriptions that were not directly relevant to this model (such as the definition of the DHCP daemon).

```
/**
 * A queue has a name, and a link to a printer.
 */

Queue = {
   name;
   printer::Printer;
}
/**
 * A list of all queues that we want to provide
 * to users of the printing service.
 */
Queues = {}::Queue;
// all elements inside this description are queues


/**
 * Definition of a printer :
 * A printer is a node with a single interface,
 * a format, a ppdFile (capabilities of the printer
 * and Zaliases for print job submissions.
 * It belongs to a given printing subnet.
 */
Printer = Node & {
    eth0::Interface;
    format;
    ppdFile;
    Zaliases = {}::String;
    printingSubnet;
} satisfying (eth0.defined and printingSubnet.defined);

/**
 * The main list of Printers
 * This list would be list to keep up-to-date
 */
Printers = {}::Printer;
// all elements inside this description are printers

/**
 * Printers are given their IP address by a
 * DHCP server on the same subnet.
 * For this table to be valid the subnet must be defined
 */
PrintingDHCPTable = DHCPTable & {
```

```
  subnet;
  // collect IP and mac of all printers on this subnet
  dhcpMappings = {
    -- = {
      ipAddress = p.ipAddress;
      macAddress = p.eth0.macAddress;
    } where
         (p from Printers,
          p.printingSubnet == subnet);
  };
} satisfying (subnet.defined);

/**
 * Definition of a print server :
 * A print server is a node with a second interface,
 * and a printing subnet it serves. Its DHCP table
 * is a printing DHCP table as defined above
 */
PrintServer = Node & {
    eth0::NamedInterface; // will be configured
                          // by the naming system
    eth1::Interface;
    printingSubnet;
    dhcpTable = PrintingDHCPTable & {
      subnet = printingSubnet;
    };
    // the queues served are all the queues serving
    // a printer on the same subnet
    queuesServed = {} where
           (q from Queues,
            q.printer.printingSubnet == subnet);
    // define the DHCP application ...
    ...
}
/**
 * Definition of all the print servers.
 * This list is the one that would have
 * to be populated by the actual values.
 * We ensure there is at least one print server
 * per subnet.
 */
PrintServers = {} satisfying
                for all s in PrintingSubnets,
                [exists] printserver,
                        (printserver.printingSubnet == s);

/**
 * LDAP Directory
 * The printing daemon installed on machines uses
 * an LDAP directory to collect information
 * on queues: mappings from queues to print servers
 * and mappings from queues to printers.
 */
```

```
LDAPDirectory = {
  // this information already exists
  queuesToPrinter = Queues;

  // and we build pairs of queue names and
  // the print server that serves them
  queuesToPrintserver = {
    -- = {
      queueName = queue.name;
      servingPrintserver = printserver;
    };
  } where
      queue from Queues,
      printserver from PrintServers,
      [printserver.queuesServed,queue].contains;
      // this ensures the queue is in the list of queues
      // served by this print server
};


/**
 * User Definition
 * A user has a name and is allowed
 * to use a set of queues.
 */

User = {
    name;
    queuesAllowed = {}::Queue;
}
/**
 * The list of all users
 */

Users = {}::User;

/**
 * Definition of Permissions
 * Permissions are used by printing daemons to
 * map a user name to the printers he is allowed
 * to use.
 */
Permissions = {
  -- = {
    username = user.name;
    // we iterate over the queues this user
    // is allowed to use and add the printer associated
    // to this queue to the list
    printersAllowed = [printer] where
                          (queue from user.queuesAllowed,
                           printer = queue.printer);
  }
} where user from Users;
```

Bibliography

[ABK+02] Paul Anderson, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, and Peter Toft. Technologies for large-scale configuration management. Technical report, The GridWeaver Project, December 2002. http://www.gridweaver.org/WP1/report1.pdf.

[ABK+03] Paul Anderson, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, Peter Toft, and Jim Paterson. Experiences and challenges of large-scale system configuration. Technical report, The GridWeaver Project, March 2003. http://www.gridweaver.org/WP2/report2.pdf.

[AGP03] Paul Anderson, Patrick Goldsack, and Jim Paterson. SmartFrog meets LCFG - autonomous reconfiguration with central policy control. In *Proceedings of the 2002 Large Installations Systems Administration (LISA) Conference*, Berkeley, CA, 2003. Usenix. To be published.

[And01] Paul Anderson. The complete guide to lcfg. Internal Document, 2001. http://www.lcfg.org/doc/lcfg-guide.pdf.

[AS02] Paul Anderson and Alastair Scobie. LCFG - the Next Generation. In *UKUUG Winter Conference*. UKUUG, 2002. http://www.lcfg.org/doc/ukuug2002.pdf.

[BKMP03] George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau, and Jim Paterson. Design of prototype models for some problems facing large-scale configuration management. Technical report, The GridWeaver Project, July 2003. http://www.gridweaver.org/WP4/report4.pdf.

[CG99] Alva Couch and M Gilfix. It's elementary my dear watson: Applying logic programming to convergent systems management processes. In *Proceedings of the 13th Large Installations Systems Administration (LISA) Conference*. LISA, 1999.

[CP02] Lionel Cons and Piotr Poznanski. Pan: A high level configuration language. In *Proceedings of the 16th Large Installations Systems Administration (LISA) Conference*. LISA, 2002.

[Edi] Edinburgh School of Informatics, EPCC and HP Labs. The GridWeaver Project. Web page.
`http://www.gridweaver.org`.

[FKNT] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke. The physiology of the grid. Web page.
`http://www.globug.org/research/papers/ogsa.pdf`.

[gt3] The Globus Toolkit 3.0. Web page.
`http://www.globus.org/toolkit/gt3-factsheet.html`.

[HP01] Matt Holgate and Will Partain. The arusha project: A framework for collaborative systems administration. In *Proceedings of the 15th Large Installations Systems Administration (LISA) Conference*. LISA, 2001.

[jbo] The JBoss Application Server. Web page.
`http://www.jboss.org/`.

[Ser] Serano Project, HP Labs, Bristol (UK). SmartFrog – Smart Framework for Object Groups. Web page.
`http://www.smartfrog.org`.

[TCF+] Steven Tuecke, Carl Czajkowski, Ian Foster, Jeffrey Frey, Steve Graham, and Carl Kesselman. Grid service specification (draft 3). Web page.
`http://www.gridforum.org/ogsi-wg/drafts/GS_Spec_draft03_2002-07-17.pdf`.

[tom] Apache Tomcat. Web page.
`http://jakarta.apache.org/tomcat/`.