The SmartFrog configuration management framework

Article *in* ACM SIGOPS Operating Systems Review · January 2009

DOI: 10.1145/1496909.1496915 · Source: DBLP

8 authors, including:

Patrick Goldsack
HP Inc.
24 PUBLICATIONS **495** CITATIONS
SEE PROFILE

Julio Guijarro
Hewlett Packard Labs
9 PUBLICATIONS **230** CITATIONS
SEE PROFILE

Steve Loughran
Apache Software Foundation
20 PUBLICATIONS **220** CITATIONS
SEE PROFILE

Alistair N. Coles
HP Inc.
11 PUBLICATIONS **170** CITATIONS
SEE PROFILE

The SmartFrog Configuration Management Framework

Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles,
Andrew Farrell, Antonio Lain, Paul Murray, Peter Toft

HP Laboratories, Filton Road, Bristol BS34 8QZ
+44-117-9799910

{firstname.lastname, pmurray}@hp.com

ABSTRACT

SmartFrog is a framework for creating configuration-driven systems. It has been designed with the express purpose of making the design, deployment and management of distributed component-based systems simpler and more robust. Over the last decade it has been the focus for ongoing research into aspects of configuration management and large-scale distributed systems, providing a platform for experimentation. The paper covers the rationale for the design of the framework, details of its design, plus a description of the further research that is in progress.

Keywords

Distributed systems, configuration, deployment, automation, lifecycle, SmartFrog, management, scalability, autonomic computing.

1. INTRODUCTION

The explosive growth of networks in the early 1990s, coupled with the deployment of new network technologies, from mobile networks to ATM backbones for the Internet, brought a need to monitor and manage networks in a way that had not previously been necessary. Although performance and status information was being collected at network devices, and specialized probes and instruments were being developed by companies such as HP, the large-scale distributed systems required to gather and analyze such information did not exist.

The few attempts at building such systems frequently crumbled in the face of their scale and complexity. Consequently a research project was started in HP Laboratories to look at the design of these systems and to develop core technologies that would enable them to be created and deployed much more rapidly. This research has progressed naturally from the field of building large monitoring systems to the space of managing large-scale infrastructure and services. As such, this work is as relevant now as it ever was.

The research goals for the development of systems management technologies were determined by trying to understand the problems that the groups building these large systems had faced, with the aim of extracting the primary reasons for them. In this process, a number of aspects became clear very early on, and significantly all were related to system configuration and lifecycle management.

On detailed examination of the software engineering practices involved it was clear that the principal causes of the system design problems lay with the ad hoc way in which such large and distributed systems are built, where individuals make their own decisions about configuration and lifecycle. This had never been identified as an important issue on par with, for instance, the detailed definition of the APIs between the various components. Configuration data was scattered throughout the system and often repeated – for example, the location of a server process might be held in multiple locations, making a change in the host running a server instance a complex task. As a result, initial configuration and particularly routine maintenance proved very hard to do. Part of the data would be forgotten or, as frequently happened, users would not change configuration data to match changes in requirements and the systems would degrade over time.

However, it was not only configuration data that was a problem – the lifecycles of the components were frequently incompatible, thus making it hard, if not near impossible, to bring a system up or down as an automated process. For example, ordering dependencies could not be met if components did not correctly sub-divide and phase their start-up or shut-down processes.

Lastly, from a configuration and lifecycle management perspective, failure and recovery was usually inconsistently detected and handled, leading to great difficulty in creating reliable self-recovering systems. In some cases this was so bad that the re-engineering of the configuration and lifecycle management piece of one system was dubbed its “high-availability program”.

Given this backdrop the research group tackled the task of providing a systems management framework that could be applied across a number of areas – not only that of the management of distributed monitoring systems.

The research concentrated on three main areas:

- How to describe configurations that take into account the needs of product development, deployment and run-time lifecycles;
- How to use this to drive automation in the lifecycle management;
- How to build scalable and reliable “configuration-driven” systems using the technologies developed.

The primary output of this work was the SmartFrog (Smart Framework for Object Groups) technology and associated components [1]. It dates back to 1996 and has been used in several products by HP and other companies. It has been in open source since 2003, and is still under active development. Here, we give a retrospective account of SmartFrog, including the rationale for how it was developed, as well as what worked and what did not. We also look to the future, giving some details of developments that have been planned for future releases in view of our past experiences.

2. SMARTFROG BASICS

In order to bring some structure to the chaos that existed, the first task was to develop a simple and basic model of configuration and lifecycle that could be used to drive the design of the underlying technology. To this end, we defined the notion of a SmartFrog

Component and defined a SmartFrog *System* to be a collection of such components.

A SmartFrog component consists of three pieces as illustrated in Figure 1:

1. Configuration data for the component, defined in a standardized way and with standardized APIs to access the configuration data.
2. A lifecycle manager that uses this data to drive the configuration of the component and also provides a standardized way of defining and transitioning the lifecycle steps of a component.
3. The functionality of the component itself which may provide any functionality-specific interfaces, using the configuration data to define detailed behaviors. This is known as the managed entity and in essence it provides the semantic interpretation of the configuration description.

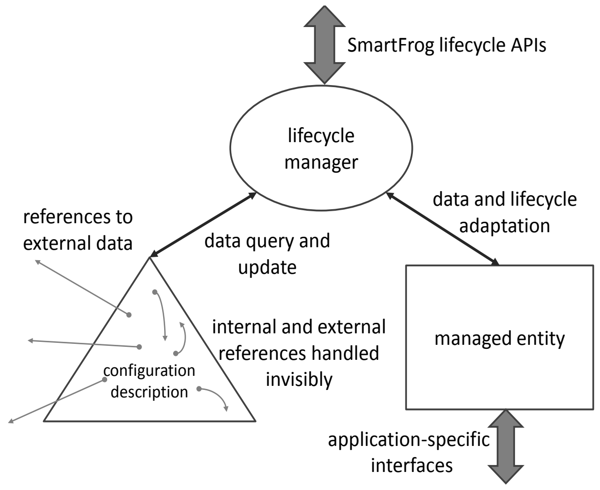
SmartFrog lifecycle APIs
lifecycle manager
references to external data
data query and update
data and lifecycle adaptation
internal and external references handled invisibly
configuration description
managed entity
application-specific interfaces

Figure 1: Component structure

Frequently the managed entity will be provided as an existing piece of software, for instance a third-party product or a component lifted from another system for reuse. It is the lifecycle manager's role to wrap this – to adapt the unified configuration data and lifecycle model to that of the managed entity including translating the configuration data into whatever format is expected by the managed entity. It is also possible to merge the notion of the lifecycle manager and the managed entity if the component is being written specifically for the SmartFrog framework.

A system is then defined to be a set of such components whose configuration data might be inter-related in some way with its overall lifecycle emerging from the combination of the individual component lifecycles. This composition of data and lifecycle is a key step in constructing large-scale systems, and the primary goal of the SmartFrog framework – the sets of components being the "object groups" of the SmartFrog name.

SmartFrog descriptions define the nature and configuration of components, but these need to be realized to create the running system. SmartFrog provides a run-time infrastructure, consisting of sets of daemons running on whichever hosts are to be included in that system. The daemons provide all the underlying support to take the descriptions and turn them into a running distributed system. This is illustrated in Figure 2.

Note that the components created are not "fire-and-forget". As part of their lifecycle they are expected to monitor and detect any failure of the managed entity and to report this to other parts of the system of which they form a part. Protocols to monitor the health and existence of a component are provided as part of the framework and this is done periodically across the system or on demand if special circumstances require it.

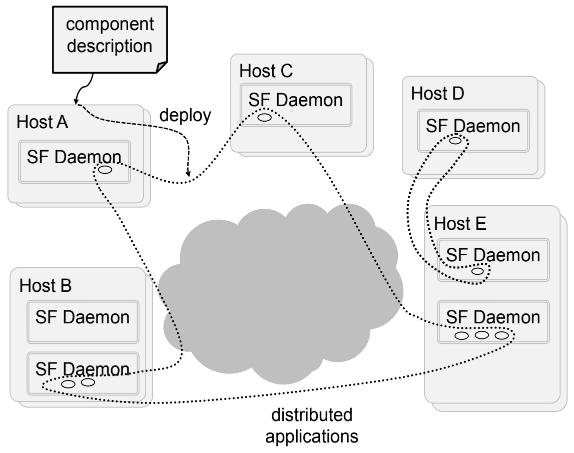

Figure 2: Runtime system

SmartFrog is implemented entirely in Java and is developed and tested on a range of server and client operating systems.

3. DESCRIBING CONFIGURATIONS

In order to achieve a more coherent and consistent configuration environment for a system, it is necessary to provide a uniform and natural way of describing configuration data for the collection of constituent components. This must be done in a way that supports a number of different aspects essential to the overall goal of specifying the shape and configuration details of a system.

Composition of descriptions is required so that the pieces of a system may be flexibly composed with others and any commonality of configuration information may be shared. This must be possible even though a specific composition may not have been envisaged ahead of time.

Configuration information may arise from many epochs in a system's life. Component writers will provide example or default configuration values and determine the overall shape of the data. Integrators will relate values across the selected components. Installers will add specific data associated with the context within which the system is installed. Finally the system will be deployed and whilst running, one part of a system may generate configuration data for another. A common example is a dynamically assigned TCP port, a port whose value will only be known when the application is started.

This leads to another essential requirement – the ability to indicate when certain parts of a configuration are not known statically, but must be discovered or generated at run-time. Late binding – lazy binding in SmartFrog terminology – of configuration data is best provided in a way that is hidden to the components using the data. SmartFrog supports the addition of new mechanisms, for example new discovery protocols or naming services, to the framework so that these can be used within the descriptions, even those for preexisting SmartFrog components. This has the advantage that component writers do not need to predict all ways in which configuration data may be obtained in a system context. This considerably improves reuse and reliability.

Finally, an essential idea in configuration is that of templates: a parameterized description of data that may be used in many contexts. In the process of composition, for example, a basic template may be used to provide core default values and provide the essential structure, but then modified as part of the composition to capture relationships between elements of the composition. This composition may in turn be a template for further composition into perhaps several larger systems. This idea that all descriptions are templates for refinement and modification is core to the way SmartFrog works.

3.1 The SmartFrog Notation and Data Model

To support the above requirements, SmartFrog contains a declarative data description notation that is defined along with supporting tools and programmatic data structures. The language may be parsed and processed to provide a queryable structure of the data model for the underlying configured component and, if necessary, other components within the system.

The core of the data model is the notion of an ordered hash table, known in the SmartFrog terminology as a *Component Description*. Since the value named by a key may itself be another component description, this structure allows for simple and natural hierarchical data with an overlaid naming structure – paths through the hierarchy similar to paths in most file systems. The SmartFrog name for a key-value pair in such a table is *attribute*. In the following example, based loosely on a snippet of a configuration file for the Apache web server, the keyword *extends* is used to introduce an attribute whose value is defined to be a component description. The following snippet defines a component description `webServer` which contains a number of semicolon-separated attributes, namely `port`, `logLevel` and `dataDir`. This last attribute is itself a component description containing attributes.

```
webServer extends {
        port 80;
        logLevel "WARN";
        dataDir extends {
                path "/usr/local/webdata";
                permissions extends {
                    order ["deny", "allow"];
                    deny "all";
                    allow path;
                  }
            }
    }
```

The data model seen by the components when looking at their own configuration data is entirely captured in this notion of hierarchical data. Indeed, when designing the lifecycle manager for a specific component, the most important part is to define this data model.

However such a simple model does not satisfy any of the requirements for configuration given above. The core value of SmartFrog is in how these data structures may be defined in such a way as to seamlessly enable the definition of templates, to late bind data, avoid data repetition, and so on. Each of these is captured within the configuration description, but not directly visible to the component.

This two-sided view of the configuration data is an essential feature: on one side a component has a simple hierarchical view of its data that it understands, but on the other the way that the SmartFrog framework actually generates that view for a specific instance at run-time must be highly flexible. In this way, a component may be used in an open-ended set of possible configurations, each defined by use of the description notation, while the component always sees the same simple hierarchical structure it requires.

This approach provides a greater level of reuse and allows components to be tested in one context (perhaps for simplicity using files to hold the configuration data), and deployed in another (perhaps requiring a database for reliability and transactionality, or perhaps run-time discovery). The SmartFrog configuration descriptions and run-time system map these appropriately for the component whose implementation does not need to change.

3.2 Composition and Extension

SmartFrog's primary mechanism for composition is the ability to include one component description as an attribute in another. In this way, configuration data for many components may be composed simply by their inclusion within a common description. The keys provided for these descriptions ensure that each part may be referenced from another, i.e. that the naming paths exist.

Extension is the core of the template concept supported by SmartFrog. Extension is introduced by the *extends* keyword and allows for the combination of a comma-separated list of component descriptions either denoted by reference to an existing description, or by providing the attributes directly within the definition. The definition of `pathAllowed` illustrates this.

```
noPerms extends {
       order ["deny", "allow"];
       deny "all";
    }
pathAllowed extends noPerms, {
       allow path;
    }
directory extends {
       path TBD; // a value yet To Be Defined
       permissions extends noPerms;
    }
webServer extends {
       port 80;
       logLevel "WARN";
       dataDir extends directory, {
               path "/usr/local/webdata";
               permissions extends pathAllowed;
           }
    }
```

The semantics of extension are illustrated in the example. A description that is extended effectively contains all its attributes with their existing values unless added to or redefined by the extending set of attributes. Many definitions and modifying sets of attributes may be given and these are applied in the order specified. The resultant definition of `webServer` is identical to that in the previous example.

3.3 References

References are used to refer from one location in the data structure to another in a way similar to that of a symbolic link present in most file systems, referring from one file to another. A reference is indicated as a series of colon-separated steps through the hierarchy of data, including referencing the containing description, the top (root) description, and any attribute by its key within a description. By concatenating these steps into a reference, any value may be indexed from any other part of a hierarchical description.

```
clientServerSystem extends {
        serverPort server:port;
        server extends webServer, {
                port 80;
           }
        clientApp1 extends {
                   port serverPort;
           }
        clientApp2 extends {
                   port serverPort;
           }
    }
```

This provides a number of important capabilities and in particular the ability to provide both abstraction of structure and the means to avoid data replication through information sharing.

The sharing is illustrated in the example above, where both of the client *serverPort* attributes share the value of the enclosing description's *serverPort* attribute, which in turn is mapped to the server's *port* attribute. The top-level definition of *serverPort* is an example of how the use of references may abstract the structure of the configuration data. This is extremely important to support versioning and backward compatibility – using references to provide multiple views of the data depending on version requirements.

3.4 Late binding and adaptation to context

Configurations may contain data that comes from different stages of the processing of a description. There may be data that is known in advance provided by the writer as part of the description. Other data may only be known much later, either during the processing of the description – parsing, resolving the extensions, following the references and evaluating the expressions representing configuration values – or at run-time as the system is running and dynamic or local data becomes available.

So that the component itself does not need to encode all possible ways to obtain configuration data, one important aspect of SmartFrog is ensuring that the description itself contains these details and the component simply requests the required attributes. This implies that the ways in which data may be located must be extensible within SmartFrog. The way in which SmartFrog encodes access to external data is through the use of intercepted references.

References are followed step-by-step; each part of the reference defines the next context for the rest of the reference to be followed, the initial context being the location of the reference. At any point, a reference resolver can be inserted into the configuration description which, when given the remaining part of the reference, can use this as a parameter to locate the data in any means available to it, returning the data in exactly the same way as a more usual reference would.

There is a need to invoke these context adaptors at different points in the lifecycle of the system, to be resolved at the time of language processing or delayed until run-time, so called late-bound or lazy data.

```
system extends {
   server1 extends webServer, {
       port LAZY ENV webport;
   server2 extends webServer, {
       port (server1:port + 1);
   }
}
```

The example shows the use of the keyword *LAZY* to indicate that the attribute *server1:port* is determined at run-time, in this case from the environmental variable *webport*. Note that data that depends on this late-bound attribute cannot be evaluated until it is available. So the late-binding of data is automatically propagated through the description as required and dependent expressions remain unevaluated until the all the data is available. In the example, *server2*'s port depends on this late-bound data, so it too becomes late-bound by propagation.

The SmartFrog system can hide the late binding from the component when required, and if the attribute value is not available on request of the configuration data, an attempt is made by the SmartFrog framework to obtain the data and evaluate any expressions required at that instant. If obtained, it is returned, almost as though it had been statically defined. Failure, however, cannot be hidden and is reported back to the component.

3.5 Validation

Configuration descriptions, and especially those that are to be used as templates and so modified through extension, should be validated against component-specific assumptions that may not be present in the structure alone. For example the port number in the configuration of a web server must certainly be less than 64k. In a more complex example, as it relates multiple attributes, a component requiring attributes that represent minimum and maximum bounds of some value presumably expects the minimum be less than the maximum.

In other notations such as XML these constraints are captured within the notion of a schema, a separate document that defines conditions for a description to be well-formed. However SmartFrog takes a slightly different approach. Rather than having parallel structures for the two concepts – schema and description – SmartFrog combines them within the same structural form by attaching these conditions as predicates on a description. When a description is extended these conditions are propagated and additional predicates may be added, semantically forming a conjunction between these old and new predicates.

```
webServer extends {
    port TBD;
    logLevel "WARN";
    valid extends Assertions, {
        portOK (port <= 0xFFFF);
        logOK (logLevel == "WARN" OR
               logLevel == "ERROR");
     }
}
system extends {
   server1 extends webServer, {
        port 80;
   }
   server2 extends webServer, {
       port (server1:port + 1);
   }
   valid extends Assertions, {
       portsDiff (server1:port != server2:port);
   }
}
```

Conditions are expressed by adding attributes whose value is a set of predicates. The name of the attribute does not matter, but it must extend the built-in component description *`Assertions`*.

Consequently, as description templates are used and refined, conditions on their use are propagated and may be strengthened. This supports a number of scenarios, not least of which is that of multiple epochs. Component developers may annotate the configuration templates they provide with restrictions on use of their components; those who refine them and combine them into systems may add further such conditions associated with the specific system context; and those who deploy and manage them in the field may add additional local policy conditions. These predicates are all defined within the template descriptions themselves and validated at appropriate moments in the lifecycle of the overall system.

4. CONFIGURATION LIFECYCLE

It is not sufficient merely to provide the data to each of the system components, it is necessary to act on the data and configure the system according to the intention represented by that data. In order to convert configuration data into the semantics of a specific configuration, SmartFrog provides the ability to attach a lifecycle manager to any piece of the configuration data – functionality intended to manage the lifecycle of the underlying managed entity whose configuration the data represents.

For example, some part of the overall configuration description may represent a web server and configuration aspects such as its port and directories which it is to serve. Attached to that data may be a SmartFrog lifecycle manager that configures and manages an Apache Server according to the configuration data to which it is attached.

In a large system there may be hundreds of components that need to be created and managed. For setup, tear-down and update it is likely that there will be ordering dependencies across the set of components. For example a server may need to be installed and started before a client attempts to access it. The requirement to be able to manage groups of components with inter-related lifecycles was a core driving requirement for the design of SmartFrog.

In order to configure a distributed system, it is necessary to ensure that the distribution of configuration data and the coordination of component lifecycles are possible seamlessly and securely, both locally and remotely. This allows configuration descriptions to represent and manage both centralized and various distributed configurations by changing a few details to do with locality. The SmartFrog framework has a number of predefined ways of describing locality and distribution, and being a framework it provides the mechanisms for adding more.

4.1 Lifecycle Managers

In order to convert configuration data into a running and properly configured system, the SmartFrog data needs to be associated with one or more lifecycle manager components – each an instance of a Java class, and known for historic reasons in SmartFrog as a *Primitive* component – to cause the right configuration actions to occur, in the right location, and in the correct order. The process of taking a component description and creating the lifecycle managers defined within it is known as *deploying* a description.

The way to describe the lifecycle manager for the configuration data may differ according to the specific setup of the SmartFrog framework but is usually through the provision of an attribute that defines the name of the class, *`sfClass`*, and an optional description specifying how to set up the class path for Java to locate that class. SmartFrog supports the dynamic downloading of classes so that the machines being configured do not to need to be pre-loaded with all possible classes.

Note that it is possible to separate out the configuration data for a specific class of application from the details of the implementation. Consider as an example the configuration of web servers. Many of the configuration parameters of a web server are similar, no matter which specific type of server is being managed. A lifecycle manager could be created for each of the web server types, such as Apache[2] or Jetty[3], and each expects a common abstraction for the configuration data. The selected type can then be combined with the configuration data using the *`extends`* operator in the SmartFrog notation.

```
webServer extends {
    port 80;
    //etc for the generic web server data
}
jetty extends {
   sfClass "org.smartfrog.jetty,Jetty";
   // etc for jetty specific data
}
apache extends {
   sfClass "org.smartfrog.apache,Apache";
   // etc for apache specific data
}
myJettyServer extends webServer, jetty;
myApacheServer extends webServer, apache;
```

Each lifecycle manager is expected to implement a specific set of methods that drive the various states of the lifecycle. This includes methods for a phased activation process consisting of creation, initialization and startup, plus termination, failure notification and status checking. The lifecycle manager may trigger termination itself, or it may be told to terminate by the framework. These lifecycle methods must be mapped by the lifecycle manager to the appropriate actions on the software or hardware that it is responsible for, its managed entity. Thus *starting* a lifecycle manager really means *starting* the managed entity for which it is responsible, whatever that means in the context of the overall system.

4.2 Composition: Compound Managers

The notion of a group of components in SmartFrog is known as a *Compound*. The key thing about groups of this kind is that they too have a lifecycle and semantics, but these are entirely about the group of components that they manage. So, for instance, creating, initializing and starting a compound may involve phasing those actions across its members. There are many possible semantics for a grouping of this kind and SmartFrog provides a number of different ones and, being a framework, new ones may be defined and added to the set.

In effect, the deployment of each compound results in the deployment of a lifecycle manager for the associated group. The configuration data for the manager is the list of members of the group. This manager implements the group semantics including the phasing of all lifecycle steps for the members.

Each of the predefined types of compound manager is associated with a predefined extensible description. For example *`Compound`* (phased startup, shared fate – one terminates, all are terminated), *`Sequence`* (sequential startup, next to be started on termination of previous, order defined by attribute order in the description), and *`Parallel`* (independent startup and fate).

```
system1 extends Compound, { //shared fate
   server1 extends webServer;
   server2 extends webServer ;
}
system2 extends Parallel, { //independent fate
   server1 extends webServer;
   server2 extends webServer ;
}
```

This example shows a pair of server components that are handled as a compound group with shared fate (one terminating will cause the other to terminate) and independent fate (each terminates independently).

Complex structures of nested group lifecycles can be rapidly described. Judicious use of Sequence and Parallel in particular gives the effect of a configuration workflow, whereas Compound tends to be used for closely coupled collections, where the existence of one component without the others makes no sense.

4.3 Run-time Structure

During the deployment of a component description the lifecycle and compound managers are created as required and driven through the phased start-up lifecycle. Each of these managers contains within it the component description that holds its configuration data. In this way, the same naming structures are maintained at runtime as were present in the original component descriptions.

This means that references defined within the component description can also be dereferenced once all the components have been created. This duality between data structure and component structure, with its consistency of naming and reference, is an essential feature of SmartFrog and underpins the semantics of delayed binding.

Any component may add or replace any of its attributes at runtime, for example providing its creation time or run-time status or some other discovered data. These attributes may then be accessed by other components within the system using lazy references. These attributes are also available to any management tools that take advantage of these runtime structures.

4.4 Location and Distribution

SmartFrog was initially conceived as a way of configuring distributed systems. Consequently one important feature included in the framework from the start was the ability to specify the location of each component within the distributed set of nodes. Location information is included as attributes with the definition of the component's configuration data to ensure that SmartFrog created the components in the right locations.

The way locations are defined may differ from system to system, and new ways can be added to the framework to cater to new situations through the use of new location managers, known as deployers. So in one system the host name may be used to determine location while in another the location may be defined by some property such as the proximity to some other part of the system.

Whenever a new component is to be created to realize some piece of the configuration data, the SmartFrog deployers inspect the data to extract location information, find the desired location and create the component there, complete with a copy of the configuration data for that component.

```
system extends Compound ,{
        server1Host TBD;
        server2Host TBD;
        server1 extends webServer ,{
                sfProcessHost server1Host;
            }
        server2 extends webServer, {
                sfProcessHost server2Host;
            }
    }
sameHostSystem extends system, {
       server2Host server1Host;
    }
systemA extends system, {
        server1Host "hosta.hpl.hp.com"
        server2Host "hostb.hpl.hp.com"
    }
systemB extends sameHostSystem, {
        server1Host "hosta.hpl.hp.com"
    }
```

The example demonstrates how SmartFrog defines locations through host names, and also demonstrates how different distribution patterns may be achieved with simple extensions of templates. This is the property that makes it simple to create templates that are usable in many contexts – for example in various testing as well as production deployments.

From the user's perspective the distribution can be largely hidden and in particular the hierarchical naming structures provided by the components are identical whether the components are fully distributed or entirely local to one machine. The SmartFrog system takes care of resolving references that cross distribution boundaries.

The SmartFrog runtime system consists of a collection of daemons, each a Java virtual machine running a single pre-defined SmartFrog component. This component manages the daemon and provides the initial access point for all operations, such as deploying descriptions and managing and locating all user-deployed components. The daemons may reside on the same host or be distributed across a number of hosts. The daemons, once started, may advertise themselves in any form required for the location mechanisms encoded into that instance of the SmartFrog framework, such as a host-local registry or a central registry service such as DNS, or a distributed discovery service such as the Service Location Protocol [4]. This specialization may be easily achieved by defining a set of SmartFrog components which are deployed according to a set of initial component descriptions.

It is the essence of the framework that such aspects of run-time support may be modified in a way that does not impact the implementation of components.

5. SYSTEM CONFIGURATION

5.1 Collections of subsystems

As the systems being created get larger and more complex, they can no longer be described and managed through a single monolithic description and related set of SmartFrog components. The system needs to be decomposed, and the distributed state managed through protocols and services for discovery and naming. SmartFrog provides a standardized way of incorporating these so that they seamlessly provide the distributed configuration data without the tight dependencies of a single description, and to provide support for partial failure and recovery.

Individual descriptions are ideally used for components whose existence within the system is mutually dependent, components that might as well share fate with each other: that are deployed and terminated as one.

However when using multiple descriptions, we do not want to lose the benefits associated with a single description such as the sharing of configuration data and obtaining late-bound information though lazy references. Furthermore, to support the many possible deployment scenarios – for example testing or live deployment – we need to be able to provide configuration descriptions that are sometimes monolithic, sometimes decomposed. However it is SmartFrog's goal to ensure that from any component's perspective, these appear identical.

For the simplest case, that where a reference is required to a component residing in a known daemon, SmartFrog has efficient built-in support for referencing it. However for more complex cases, for example where a client has no idea where a server resides, discovery protocols or naming services must be integrated.

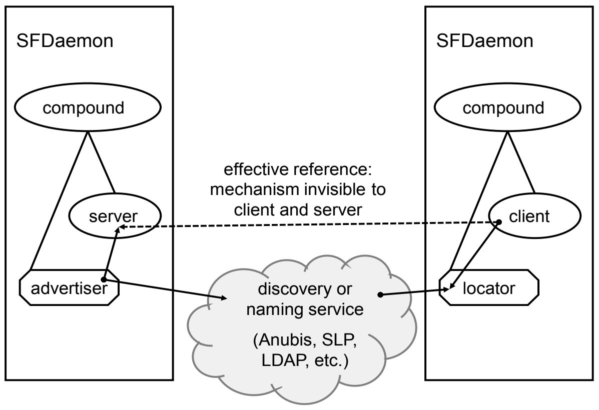
SFDaemon
compound
server
advertiser
effective reference: mechanism invisible to client and server
discovery or naming service
(Anubis, SLP, LDAP, etc.)
SFDaemon
compound
client
locator

Figure 3: Discovery and naming pattern

Discovery or naming adaptor components are added to a description to provide the appropriate model of the system, hiding the separation. There are two types of adaptor – an advertiser that export the whereabouts of a component, and a locator that will find it and intercept the references and map them appropriately. This is a standard pattern that is used for integrating all naming and discovery services and is illustrated in Figure 3. Note that in this example, neither the client nor the server need be aware of the service being used.

SmartFrog comes packaged with a number of such services – the Service Location Protocol being one such. However the primary protocol used in-house is one which has been specifically written for SmartFrog, called Anubis [5]. This protocol has a number of features, all of which are useful in the context of systems construction. It provides a group membership protocol, with strong timing and view consistency properties. It also provides a state distribution service, so SmartFrog components may advertise and exchange service locations and configuration data.

5.2 System Adaptation

Systems change over time, sometimes over long periods, sometimes over relatively short periods, and especially if there is the need for dynamic adaptation to changing circumstances. These changes may be to add or remove system components to cope with demand – for example adding instances of web servers to cope with peaks in demand – or they may be to cope with recovery after failure. To handle these cases autonomically there is the need for programs to interact with the SmartFrog system to dynamically create subsystems and have them deployed by SmartFrog. To this end, all parts of the SmartFrog framework are accessible through APIs – from creating and processing of new configuration descriptions, to giving these to SmartFrog to be deployed. Running subsystems may be terminated, inspected and modified. In this way it becomes very simple to develop and deploy highly adaptive systems. An example is described in Section 7.

5.3 Security

Given that SmartFrog is an automated distributed configuration management system, it is open to attack. Attackers could attempt to use SmartFrog to deploy unwanted software and components or to interact with the software that is there to terminate it or modify its configuration.

Consequently it was an essential requirement for the development of SmartFrog to incorporate a rigorous security system from the ground up.

There are a number of threats that need to be removed. The first of these is the ability to request SmartFrog to deploy a configuration description erroneously, or to cause it to deploy a modified or hostile description. The second is the need to ensure that the dynamic class loading does not cause classes that have been tampered with to be loaded in place of the correct ones, even if the classes are downloaded from an untrusted site. Third, we want to ensure that the management communications between SmartFrog daemons are appropriately secured.

We have experimented with a number of security models, including some rich and sophisticated models that have been published elsewhere [6], but in the end we decided that a clean and clear model was more important than a highly complex solution that would be hard to work with. We did not want to put unnecessary obstacles in the way of users understanding, configuring and using the security features.

The model is based around the notion of a security domain – a number of daemons may be within that domain and if so, are fully trusted by the other members. Daemons obtain security credentials from a common certification authority (CA) that defines the domain. These credentials are used to establish mutually authenticated secure channels (using SSL) for all management communication. In addition all code and descriptions which reside outside of that domain for dynamic loading must be obtained from packaged files that have been signed by the CA. The security implications of running that code are up the developer and SmartFrog clearly cannot assure that the system remains secure if the code deployed exposes the system to attack.

The principle is that the borders of the domain are heavily protected, but once inside the system behaves as normal. However to limit the damage that a single penetration into a domain can cause, it is recommended that a system be defined as a set of domains, and mediating components written to provide whatever policies are required between these domains.

5.4 Scalability

Clearly, the way in which the framework scales, both in the number of distributed nodes which it supports and the maximum size of a single description and how it is deployed, will determine its underlying utility. In this respect SmartFrog has some interesting properties.

The first point to note is that SmartFrog is passive unless called upon to carry out some specific action. A daemon, if unpopulated by components, does not communicate with others and so there is no inherent limit to the number of nodes within a single system. The limitations are rather in the way SmartFrog is used.

The next place for limitations is in the size of a single description. The system has quite happily dealt with descriptions which contain over two thousand component descriptions which deploy to a similar number of run-time components. In these tests, the managed entities were non-existent, but it is unlikely that the maximum number of components within a single description will be a serious limiting factor. More problematical might be the way in which a description for a distributed set of components may get created and managed. Since SmartFrog is fundamentally a peer-to-peer system which devolves deployment actions to the compounds for groups of components, this can happen in parallel across the structure and so poses few real limitations.

In any case, it is not recommended practice to have descriptions which are overly-large with very large numbers of components tied into a single application. Preferably, the system should be broken up into smaller parts and then reconstructed using discovery and naming protocols with references providing the abstractions for the configuration data. Under these circumstances, and since the deployment of the piece parts is not dependent on any central service or distributed coordination, the framework can scale very well.

Limitations are most often found in the specific uses of SmartFrog, for example, a specific protocol such as Anubis (to about 500 daemons) or a system architectural choice such as the use of a central configuration database. These are of course not intrinsic to the implementation of SmartFrog but are design choices about its use.

6. OPEN SOURCE

In 2003 it was decided to release the SmartFrog framework to open source. The primary reason for this was to enable and support joint research with academic partners in the configuration management space, including CERN OpenLab for Grid configuration and the LCFG team at Edinburgh University [7] which included joint integration experiments between the two systems [8]. A further consequence of the release is that SmartFrog became the focus of a standardization activity within the Open Grid Forum as the “Configuration Description, Deployment and Lifecycle Management Working Group” [9].

The open source release consists of the core framework and a host of pre-defined templates and lifecycle managers for 3rd party software such as Hadoop and JBoss. In particular there is a collection of components provided to configure and deploy three-tier web applications, especially those deployed in Java application servers. Every operation from copying files, creating database tables, and deploying web applications to probing web pages for health is supported.

The flexibility of the configuration descriptors to describe multiple different deployments of components, plus the automation of those deployments, makes it very well-suited to the automation of the test processes for large-scale systems. Add to this the automated configuration and creation of virtual machines within which the deployments may be done, and SmartFrog can be seen as a very good match for the development of test labs. SmartFrog is supplied with components to deploy test configurations, collect the results and report on whether the tests are successful. Indeed SmartFrog is used to automatically test itself. Whenever changes are committed to the code base, a stable version deploys daemons for the modified version and runs a set of regression tests over this with any failures being notified back to the developers.

7. USE AND EXPERIMENTATION

An illustration of the use of SmartFrog to configure and manage a complex software system is found in the HP Labs SE3D [10] program from 2004/5. SE3D was an HP Labs experiment bringing together a range of HP Labs research technologies. These were used to create a production-capable automated compute utility which was used to deliver a remote utility computing service for 3D animated film rendering. Twelve teams of professional film-makers each produced a 5-minute animated film using the SE3D service for the frame-rendering component of the production cycle. The technologies in the experiment included SmartFrog for system configuration and automation; an auction-based automated market system for dynamically allocating resources between the film-making teams; and compression/synchronization technologies for transferring data between the film-makers’ client PCs and the remote rendering service with high efficiency.

7.1 SE3D System Architecture

The server side of the SE3D system ran on a cluster of approximately 120 dual-CPU servers. The SE3D system software was composed of two primary layers:

1. A **resource management layer**, responsible for managing physical resources – i.e., servers – and dynamically allocating them across a population of service instances. The resource layer provided interfaces to control the lifecycle of services, and to manage the allocation of resources to those services. In particular, the resource layer utilized a resource schedule generated from the results of the online auctions, and reallocated resources across the set of services on an hourly basis to match this schedule.
2. A **service management layer**, supporting a dynamic set of service instances. In the case of SE3D we ran 12 separate rendering service instances, one for each film-making team, using the separation between services to isolate the work of each team.

7.2 The Use of SmartFrog in SE3D

SmartFrog descriptions were used to describe the complete configuration of the software systems making up both the resource and service layers. This included the various software components, their individual configuration parameters, where in the system each component was to run, how components were connected together, and in what sequence they were started. The SmartFrog runtime system was then used to deploy and manage the entire system, using the SmartFrog descriptions.

The use of SmartFrog in this context highlights several interesting properties:

1. **Simple, correct system integration:** the software system was described as a set of subsystems, each captured in one or more SmartFrog templates. Typically, each subsystem was developed by a separate research team. By using SmartFrog it was easy to compose and integrate the subsystems into a

complete system and to be confident that the configuration as a whole was correct.

2. **Easy reconfiguration:** the use of SmartFrog templates allowed many different configurations to be created easily – for example we had several test configurations and several production configurations. In addition, it was easy to include configuration data in the descriptions that was discovered only at runtime, by linking to running components or extracting parameters from a configuration database.

3. **Fully automated deployment:** the SmartFrog runtime took care of the complete task of bringing up the software system, observing any required sequencing (e.g., database component running before components that need to connect to it). It was also simple to shut down the entire running system gracefully. This allowed us to perform a weekly maintenance cycle for the complete system in under an hour, including shutting the whole system down, upgrading software components or performing other maintenance, and restarting the system.

4. **Dynamic resource management:** resources were dynamically allocated to service instances, with the configuration of each resource being instance-dependent and not defined until the point of allocation. On allocating each resource, the resource manager would deploy the appropriate SmartFrog description to the resource, which would then be auto-discovered by the relevant service instance.

7.3 SE3D Results

Twelve complete animated films were made using the SE3D utility rendering service, many of which went on to win awards and feature prominently in international film festivals. In creating these films, the rendering service ran for 10 months, providing 500,000 hours of CPU while rendering 500,000 frames. The SE3D service ran with better than 99% reliability with SmartFrog handling the autonomic adaptation to cope with the failure of hardware or software on a server. There were a total of about 170 such events over the lifetime of the system, most of which were recovered by simply restarting and reconfiguring the affected server.

Building and operating the system allowed us to improve our understanding of the performance of the research technologies in real-world applications offered as a service.

8. FUTURE DIRECTIONS

Although much of the SmartFrog system has stood the tests of both time and widespread usage, and indeed has been refined and developed since its initial conception to cater to new requirements, the development of SmartFrog is still continuing with research and development in a number of areas. These include basic improvements to the SmartFrog notation and new capabilities in the space of configuration data description and the development of new areas such as distributed automation.

Our target implementation technology, Java, has changed significantly since SmartFrog's initial release to open source, providing new features and capabilities that could usefully be incorporated. The most important of these is the introduction of the collection classes and related control structures. These can now be used as the basis of the run-time and compile-time data structures representing the configuration data. A new version of SmartFrog to incorporate this and other Java capabilities is in progress.

Lessons learnt from the long-term use of the SmartFrog notation and structures include identifying where people have had conceptual difficulties, where there is a lack of natural expressive power, and common usage patterns that could be better-supported. These all indicate that there are a small number of areas where the existing notation falls short of the ideal in both syntax and semantics. Consequently an effort is under way to revise the language to provide more rigorously defined semantics and greater expressive power.

In the early days, the design focus for SmartFrog was less around providing a rich notation, but more around creating a framework for tying together multiple data definition mechanisms and technologies into a coherent whole, and presenting these in a consistent way to system components. Consequently it was imagined that complex relationships within the data would be handled by SmartFrog's ability to add specialized functions to the framework that may then be used seamlessly within attribute definitions. However in reality there is a need to be able to represent a richer set of relationships directly within the description notation and not rely on escaping to Java to capture them.

As part of this enrichment we are exploring the ability to complete partial data descriptions whilst satisfying some specified criteria, or optimizing descriptions against some goal. The use of constraint satisfaction as part of the description allows the writer of a configuration to provide properties of collections of values rather than every individual value. This enables partial configurations to be completed based upon these properties [11]. Consequently different parameterizations of such models would cause the constraints specified to be evaluated in different ways, leading to differently completed models.

The experimental constraint extensions used by SmartFrog allow the specification of constraints such as equalities and inequalities over domain variables which have finite domains, namely subsets of integers, Booleans or enumerations. Currently the research is using the Eclipse logic programming engine [12] as the interpreter used for processing constraints. Interestingly we have discovered that SmartFrog provides an expressive higher-level syntax for constraint specification as compared with Eclipse, providing for concepts such as template definitions and inheritance which help to ease the process of domain modeling.

In the area of component and system lifecycle a number of aspects need to be developed further. In SmartFrog we defined a simple lifecycle with phased start-up, termination, and failure handling and this proved to be adequate for many situations. However, we discovered that the SmartFrog lifecycle was being used to start and stop lifecycle managers as intended but that these were then underpinning a richer lifecycle better suited to the needs of the software being managed. The components had such a varied set of lifecycles, and these were so richly interrelated, that no single lifecycle model was sufficient. Consequently, we have been working on an orchestration framework which integrates with SmartFrog to provide an easy and consistent way for these composite lifecycles to be modeled.

The orchestration framework enables arbitrarily fine-grained lifecycle management within components and provides a mechanism whereby the actions that a component may perform are predicated on the states of other components. That is to say, dependencies on the attribute state of other components may be specified for a particular component, and these dependencies determine what actions that component may take and how it may change its own attribute state.

We plan to further develop this work and are exploring its application to key areas of interest such as change management of massively large-scale, service-oriented data centers.

9. RELATED WORK

The configuration management space is well-occupied by a range of technologies but few of the major systems cover quite the same space as SmartFrog. The largest category of systems designed to manage computing infrastructure and services is based on a client-server model, where a server holds the authoritative configuration of each client and the client is maintained to match that configuration. They are "desired-state" in the sense that the descriptions of configuration provided are typically in terms of the end-state required for a client – for example the list of files that should be present on that client.

In the majority of these systems there is little or no cross-client coordination and they treat each client as an independent entity. In that sense they are weaker than SmartFrog with its ability to coordinate and configure across a range of nodes and carry out autonomic actions. However within the space where these tools are targeted, largely in desktop and device management, they do provide much more value out-of-the-box along with a simpler more direct configuration model.

Some examples of configuration management tools of this type are: LCFG [7] and its derivative Quattor [13], CfEngine [14], Puppet [15], BCFG [16], and HP's commercial *Client Automation Enterprise*, also known as Radia [17]. The precise capabilities and design criteria vary from system to system, but they are an interesting counterpoint to the SmartFrog system and the combination of SmartFrog with the client-server approach is interesting.

In a joint project between Edinburgh University, the Edinburgh Parallel Computing Centre, and us, SmartFrog and LCFG were combined to explore the benefits of such integration in more detail [8]. This work was partially funded by the UK eScience program.

There are some research contributions – Podim, [18], Alloy [19] and Quartermaster [22] – that have sought to remove the manual translation step between high-level configuration requirements and the allocation of machines to specific roles. It is through our work on constraints that we are addressing this issue. Constraint annotations in SmartFrog models specify how individual configurations should be elaborated based on the high-level configuration requirements.

SmartFrog does not include any vocabulary for describing IT artifacts such as nodes, storage or software components. A fundamental reason for this is that there are a number of initiatives, such as CIM [20] and SML [21], which have already sought to address this point.

10. ACKNOWLEDGMENTS

We are indebted to Marc Nijdam, who was involved in the earliest days of SmartFrog's conception and who wrote much of the code in the first internal version; John Manley, our Lab Director who has supported and encouraged the work throughout its life; and of course all our very many users and supporters.

11. REFERENCES

[1] SmartFrog at: http://www.smartfrog.org

[2] Apache Http Server at: http://httpd.apache.org

[3] Jetty Web Server at: http://www.mortbay.org

[4] Service Location Protocol (svrloc) at: http://www.ietf.org/html.charters/OLD/svrloc-charter.html

[5] Murray, P., 2005 A Distributed State Monitoring Service for Adaptive Application Management. International Conference on Dependable Systems and Networks (DSN-05), June 2005

[6] Lain, A., Mowbray, M. 2006 Distributed authorization using delegation with acyclic paths. Proc. 19th IEEE Computer Security and Foundations Workshop, Venice, 257–269. July 2006.

[7] Anderson,P. Scobie, A. 2002 LCFG: The Next Generation. UKUUG Winter Conference.

[8] Anderson, P., Goldsack, P., Paterson. J. 2003 SmartFrog Meets LCFG. LISA '03: Proc. of the 17th USENIX conference on System administration, 213–222

[9] Global Grid Forum. CDDLM- Configuration, Deployment and Lifecycle Management of grid services. http://forge.gridforum.org/projects/cddlm-wg.

[10] SE3D at: http://www.hpl.hp.com/SE3D

[11] The SmartFrog Constraint Extensions at http://smartfrog.org

[12] Apt, K., Wallace, M., 2007 Constraint Logic Programming using ECLIPSE. Cambridge University Press

[13] Quattor – System Administration Toolsuite, at: http://quattor.web.cern.ch/quattor.

[14] Cfengine at: http://www.cfengine.org.

[15] Kanies, L., 2006 PUPPET: Next generation Configuration Management. jLOGIN: Feb 2006.

[16] Desai, N., Lusk, A., Bradshaw, R., Evard, R. 2003 BCFG: A Configuration Management Tool for Heterogeneous Environments. Proc. of Fifth IEEE International Conference on Cluster Computing 500–503 Dec. 2003.

[17] HP Client Automation Enterprise at: http://www.hp.com.

[18] Delaet, T., Joosen, W. 2007 PoDIM: A Language for High-level Configuration Management. LISA'07: Proc. of the 21st conference on Large Installation System Administration Conference, 1–13

[19] Narain, S. 2005 Network Configuration Management via Model Finding. LISA '05: Proc. of the 19th conference on Large Installation System Administration Conference. 155–168

[20] Common Information Model (CIM) Standards at: http://www.dmtf.org/standards/cim/.

[21] Service Modeling Language (SML) Working Group at: http://www.w3.org/XML/SML/.

[22] Hinrichs, T., et al. 2004 Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation. Proc. of the 15th IFIP/IEEE Distributed Systems: Operations and Management, Nov. 2004.