SmartFrog: Configuration and Automatic Ignition of Distributed Applications

Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, Peter Toft
HP Labs, Bristol, UK

May 29, 2003

This paper provides a general overview of SmartFrog – "Smart Framework for Object Groups" – a framework for describing, deploying, igniting and managing distributed applications. The SmartFrog framework consists of a description language for specifying the configuration of applications, a deployment infrastructure for realizing the application descriptions, a component model to manage the applications through their life cycle, and a set of useful components that populate the framework to support various forms of application behavior.

Introduction

SmartFrog was conceived to overcome the myriad issues related to incorrect configuration of complex, distributed systems.

Applications such as multi-tier web-service systems are composed of multiple software components, typically running across multiple nodes. Several elements must be in place for the system to be considered correctly configured: first, each component of the system (e.g. database, application server) must be configured correctly with the required software settings. Secondly, each system component must be correctly bound to the other system components – e.g., the application server must be connected to the correct database. Thirdly, system components must be started in an appropriate sequence, such as starting the database before the application server. In addition, there are often setup tasks that must precede the startup of the main system, and cleanup tasks that must occur after the system is shut down.

Incorrect configuration in one or more of these aspects is a very frequent source of system malfunction [1]. In general, such configuration aspects are found in multiple formats, in multiple places. Changes to configurations are made one-by-one, in many places in the system, without regard for how configuration parameters are interdependent. Configuration changes are made in an ad-hoc fashion, by multiple people at multiple times (e.g., setting factory defaults, making installation-time changes, making operation-time changes), and there is little traceability or repeatability.

So, the problem is firstly how to express all of the configuration information in a consistent fashion, which recognizes the relationships between configuration parameters, avoids duplication, allows for correctness-checking, and allows the system configuration to be changed at different stages in its evolution. Secondly, how to take this more powerful expression of the system configuration and use it to automatically, repeatably and flexibly ignite the complete system.

SmartFrog addresses these problems by offering a powerful and consistent way to capture system configurations, an automatic deployment infrastructure which can repeatably realize the systems described and a component model to manage the deployed applications though their lifecycle.

The authors are members of the *Serrano* project team at HP Labs in Bristol, UK. The team's focus is on system configuration, description and ignition for utility computing. SmartFrog framework web-site is at: http://www.smartfrog.org

This document is localized for US English, A4-size paper. Published in HP OVUA 2003.

The SmartFrog Framework

In this paper, we introduce SmartFrog, a framework for the development of configuration-driven systems, design to offer a solution to these problems.

The SmartFrog Framework defines systems as collections of components. These collections of components collaborate to achieve a goal for witch they must be organized. The framework will create the components as required by the system specification; it will then correctly initialize the components with the appropriate attributes. The framework will locate and interconnect the components with each other and with different applications as required. Components can exchange information regarding their state using the framework services. We call these organizations of components *application configurations*.

We describe the SmartFrog specification language, and show its use in describing collections of software components that comprise distributed applications. We illustrate how the language supports the specification of configuration parameters for individual software components, as well as supporting the description of how components relate to one another. The latter can include a specification of the sequence in which components must be started and stopped, and a description of how components discover and bind to each other at runtime.

We discuss the design of the SmartFrog deployment infrastructure, a *Java*-based, fully distributed network of co-operating daemons that interpret SmartFrog descriptions in order to automatically and correctly instantiate the applications they describe.

We present the SmartFrog component model and how it manages the deployed systems through their lifecycles.

We discuss the security guarantees offered by the deployment infrastructure and the tools provided for managing running applications.

SmartFrog Language

The SmartFrog description language allows one to create a declarative description of the system that is wanted. The description includes such things as which software components are part of the system, their configuration parameters, and how they should connect to other components in the system, and the workflow associated with the lifecycle of the components and the system as a whole.

There are relatively few aspects of the notation that are specific to the framework. The language is used to describe attributes that the framework uses to achieve the desired configuration effect.

A description consists of an ordered collection of attributes. Each attribute has a name and a value, this value being either a basic value (integer, string, etc), or an ordered collection of attributes known as a component description. This recursion provides a tree of attributes, the leaves of which are the basic values. A value may also be provided by a *reference* to another attribute.

A *component description* consist of two parts, a reference to another component description, indicated with the keyword *extends*, to act as source of attributes, and a collection of attributes that are then added to, or override, the attributes o the referenced collection. The component descriptions may be interpreted by the framework as the description of the component or may be used to describe structured data.

The SmartFrog language is a prototype-based language which supports templates. The prototype approach makes it very easy for system configurations to be specialized for a specific context, without losing the default configuration, nor, indeed any other changes in the chain of modifications. The Template mechanism also supports this extension mechanism; in addition, templates allow multiple configuration descriptions to be composed into one. For example, this would allow a database description to be composed together with application server and web-server descriptions, and then parameterized appropriately to create a system description of a full three-tier web-service application. A simple example is shown next.

```
//webservertemplate.sf
webServerTemplate extends {
    sfProcessHost  "localhost";
    port   80;
    useDB;
}
```

```
//dbtemplate.sf
dbTemplate extends {
    userTable extends {
        columns  4;
        rows  3;
    }
    dataTable extends {
        columns  2;
        rows  5;
    }
}
```

```
system extends {
    ws1 extends webServerTemplate {
        sfProcessHost   "15.144.59.34";
    }
    ws2 extends webServerTemplate {
        sfProcessHost   "15.144.59.64";
        port   8088;
        type   "backup";
    }
    db extends dbTemplate {
        userTable:rows   6;
    }
}
```

In this example there are two component descriptions *webServiceTemplate* and *dbTeplate* that are defined as a collection of attributes. *system* is a component description with a collection of attributes that also contain other attributes. The component descriptions *ws1* and *ws2*, contained in *system*, override and customize the value of the attribute *sfProcessHost* and *db* overrides the value of the attribute rows contained in the component description *userTable* that it is included in *dbTemplate*. *Ws2* adds a new attribute-value pair, *type*.

The previous example clearly shows the two kinds of relationships between component descriptions, the containment relationship where a component description contains an attribute that is itself a component description and the inheritance or extension relationship.

Whilst the extension relationship is merely a convenient way or defining attributes, the containment hierarchy effectively provides a naming scheme by which attributes may be referenced. Ex. *userTable:rows*.

SmartFrog does not define types for attributes and components. Rather it defines the notion of a prototype. Each attribute whose value is a component description can be considered as a prototype for another: it may be taken and modified as appropriate to provide the value for the new attribute. This is done through the *extends* construct. There are no separate spaces of types and instances; every component is logically an instance, but may also be a prototype for another. An attribute may be further modified by subsequent attributes. In this way, it is possible to provide partial definitions with default values for attributes, to be completed or specialized when used. Prototyping also allows keeping all the transformation and history of configurations of the system.

Essential part of the language are *references*. Through references SmartFrog provides a flexible variable linking and component binding. References may occur in three places in the syntax: as the name of an attribute – known as placement, as a reference to the extended component (the prototype) of a component description, and as an attribute value referring to another attribute whose value is to be copied – known as link. A reference is defined in the language as a colon-separated list of parts each of which indicates a step in the path through the containment tree. The reference is evaluated in a context (a component description somewhere in the description containment tree), and each evaluation step moves the context to a possibly different component for the remainder of the reference to be evaluated.

A link reference tagged with *LAZY* will be resolved at run-time and the value of the attribute will be a link to the real instantiated object.

In the example shown next, the attribute *port* in *ws1* and *w2* is a reference to the name of the attribute *commonPort* and will be replaced before deployment with a copy of the value of the attribute *commonPort*. On the other hand, the attribute *useDB* will be resolved, once *system* has been deployed, with a run-time reference to the component *db*, not with a copy of the *db* component.

```
// List of templates
# include "webservertemplate.sf";
# include "dbtemplate.sf";

system extends {
     commonPort "8080";
      ws1 extends webServerTemplate {
          sfProcessHost   "15.144.59.34";
          port     ATTRIB commonPort;
           useDB LAZY ATTRIB db;
       }
      ws2 extends webServerTemplate {
          sfProcessHost   "15.144.59.64";
          port     ATTRIB PARENT:commonPort;
          type   "backup";
       }
      db extends dbTemplate {
          userTable:rows    6;
      }
}
```

Finally, the SmartFrog parser provides users with a small number of pre-defined functions and the mechanisms by which users may add their own functions to improve the expressiveness of the descriptions. As an example, the *concat* function is shown next.

```
// Class implementing transformation function: concat
concat extends {
   phase.function "com.hp.SmartFrog.Parser.Functions.concatenate";
}

system extends {
        server "serrano.hpl.hp.com";
        file "config.sf";

        url extends concat {
                - "http://";
                - ATTRIB server;
                - "/";
                - ATTRIB file;
        }
}
```

Transforms into:

```
system extends {
        server "serrano.hpl.hp.com";
        file "config.sf";
        url   "http://serrano.hpl.hp.com/config.sf";
 }
```

Component Model

The SmartFrog component model in no way depends on the nature of the language described above.

SmartFrog considers a whole system to be defined as a collection of applications running over a distributed collection of compute resources. This collection of applications may be dynamic, generated on demand by a variety of external or internal events, such as a user request or a new resource being started.

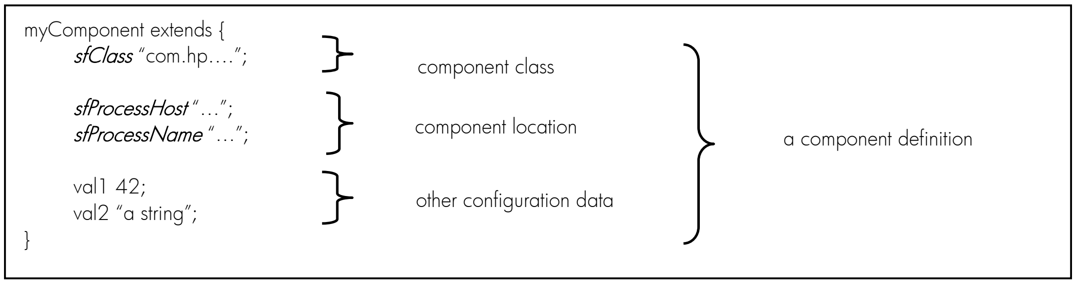
Each application is, in turn, a collection of components defined statically via an application description or generated dynamically at run-time according to the requirements determined at that time. The components of an application my be dynamic, changing over time to adjust for circumstances.

A component is defined as a single java object that implements a specific API (Prim) and which consequently implements the specific lifecycle as defined by the SmartFrog component model. The component may create and manage other objects including other processes and programs written in other languages.

An application consists of a collection of components. In an application the components are tightly bound to the others via a parent-child relationship and their lifecycles. The parents are responsible for the lifecycle of their children and are notified of child death. The order of start-up and termination is well defined. Components of an application may locate each other using the built-in SmartFrog naming capabilities.

A system is simply a collection of applications, loosely grouped over the distributed resources. Typically applications can locate each other through naming or discovery services and they must be able to cope appropriately with the non-existence of applications on which they depend, both at start-up and during operation.

Using the SmartFrog language, a component description is given to the framework to create and manage a running component associated with that description. Each description that represents a component contains to types of attributes. The first type are template attributes that define the component code, component location, and certain other management aspects. All template attributes start with the letters *sf*. The second type are component configuration attributes containing configuration information to control the component behaviour. The interpretation of these last type of attributes is component specific. A schematic diagram of a SmartFrog component definition is shown next.



```
myComponent extends {
        sfClass "com.hp....";

        sfProcessHost "...";
        sfProcessName "...";

        val1 42;
        val2 "a string";
}
```

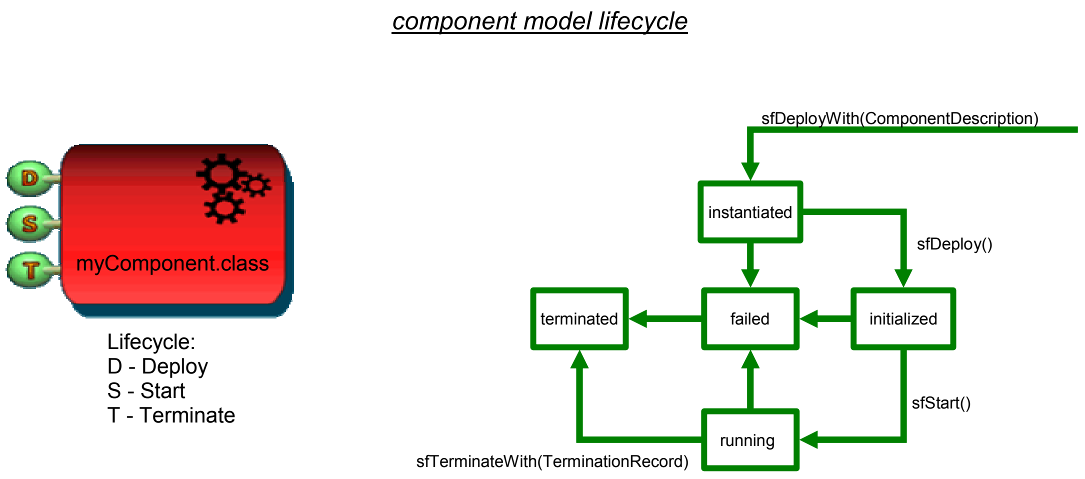
An important aspect of the SmartFrog component model is the lifecycle. The lifecycle is implemented as a simple state machine.

The transitions in the state machine are associated with actions implemented (if required) by the components. The transition actions are implemented by the invocation of methods on the component, during which the component may take any appropriate action. These transition methods, also known as the template methods, are indicated along side the transition in the diagram shown in the next page.

In an application with multiple components, the lifecycle of the whole system is defined by the combination of component lifecycles in some order. It is not completely defined within SmartFrog as to how these lifecycles are composed as it depends on the specific components used, and specifically those that are parent components.

In the next page a picture with the component model lifecycle state machine is shown.

The root parent of the parent-child hierarchy is controlled by the framework, and it is responsible for triggering the lifecycle transitions. It is the responsibility of that component to transition each of its children. The most common component used as root, and as intermediate nodes of the hierarchy is the *Compound* and this defines a simple combined lifecycle aimed at providing the notion of a single atomic composition with a shared lifecycle:



on transition to the *created* state, all children are similarly created;
on transition to the *initialized* state, all children are initialized;
on transition to *running*, all children are started;
on transition to *terminated*, all children and the parent are terminated, and thereby the whole hierarchy is terminated.

This specific semantics has some important properties. The entire application is stepped through the lifecycle in a synchronous way: all components are created before any are initialized and all components are initialized before any are started. Termination is rather different as its occurrence is asynchronous with respect to the other transitions and any component in the hierarchy may be the first to transition to the terminated state (i.e. unlike the other transitions, it is not only the root which may initiate it). Nevertheless, the semantics are such that the whole application will terminate if any component terminates.

There are other possible semantics for parent-child lifecycle combination other than *Compound*, and some are provided by the framework. In particular, the workflow package provides a number of combinations such as parallel composition and sequential composition. The workflow package is designed to provide a simple lightweight workflow-style capability to the SmartFrog framework.

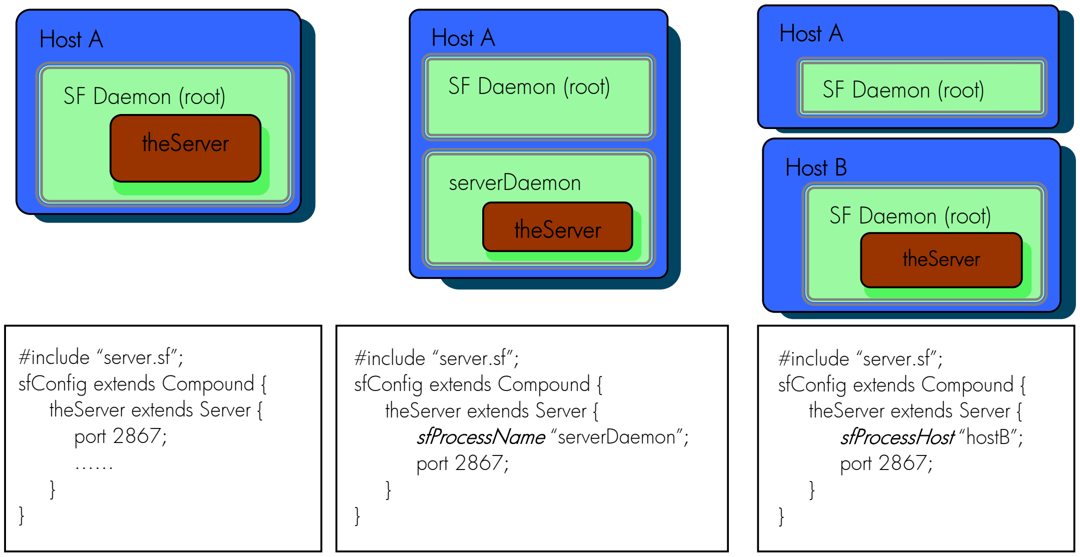
Deployment Infrastructure

The SmartFrog deployment infrastructure is a distributed network of components that interprets system descriptions, realizes the systems' subcomponents in the correct order, and binds them together. The deployment infrastructure continues to manage the components while the system is running, and is also responsible for the clean, properly sequenced shut-down of the system. If any component fails, the deployment infrastructure can detect this and can be configured to take restorative action, or to shut down the system cleanly.

The SmartFrog framework is designed to form the basis for a fully distributed configuration and programming environment. As such, the system must is able to deal with deploying components into many Processes (Java Virtual Machines or JVM) on many different hosts. The deployment infrastructure uses RMI as communication and advertising mechanism, though it is designed so that it can be used or replaced with other mechanisms.

Components are instantiated in *Processes* or JVMs. Processes come in three main types: *Root Processes*, also called *SmartFrog Daemons*, responsible for creating and managing all top-level components and subprocesses. SmartFrog Daemons are also responsible for the initial analysis and deployment of system/application descriptions. *Subprocesses* that are processes registered with a Root Process and responsible for deploying and managing components. *Basic Processes* that are Processes that do not use the core SmartFrog mechanism to advertise their existence – either because they are not required to be accessed or because they use other mechanisms for this purpose.

SmartFrog allows the user to deploy components in multiple ways, without having to change the components' code. A component is deployed by default in the root process where its application description is deployed. If the attribute *sfProcessName* is added to the component description, then the component will be deployed in a Subprocess named in by the value of that attribute. The Subprocess will be located in the same host as the default root process. If the attribute *sfProcessHost* is included in its description then the component will be deployed in a Process or Subprocess in the host named in the value. The flexible deployment environment allows the user to decide during deployment where the different parts of an application will be physically located. This behavior is shown in the next picture.



```
#include "server.sf";
sfConfig extends Compound {
    theServer extends Server {
        port 2867;
        ......
    }
}
```

```
#include "server.sf";
sfConfig extends Compound {
    theServer extends Server {
        sfProcessName "serverDaemon";
        port 2867;
    }
}
```

```
#include "server.sf";
sfConfig extends Compound {
    theServer extends Server {
        sfProcessHost "hostB";
        port 2867;
    }
}
```

Any process can dynamically load resources (such as java code) when needed for the deployment of a SmartFrog application from a central repository such as a web server.

Security Model

SmartFrog could be an extremely powerful tool to spread viruses. The ease of deploying and managing the life cycle of a distributed application plays against you when that application is malign. Therefore, it is critical that the system controls who can deploy what and where.

In particular, the communication channels between SmartFrog daemons are not necessarily secure, e.g., they could be Internet connections. An attacker could modify deployment configurations sent from legitimate daemons, or just pretend that is a valid participant and send its own. In addition, he can obtain critical configuration information, such as passwords, by snooping on the communication, that he can later use to attack the system. An important feature of SmartFrog is to dynamically load resources from web servers while deploying. These resources could be additional configuration descriptions, java classes, scripts, executable files and so on. New exploits on web servers appear every month, and the concern is that these could be used to modify resources that SmartFrog will download later on.

To help clarify the target security model in SmartFrog, the concept of a SmartFrog Trusted

Community (SFTC) has been introduced. A SFTC is a set of principals, typically composed of SmartFrog daemons and administrators that fully trust each other, i.e., they will do anything that another valid member requests, and they do not trust anybody else. There is a single authority that defines who is initially in the community, but a current member could later on add new members based on its own criteria. A member of an SFTC should only use resources that are trusted, where trust in this context means that they were created or authorized by another member of the community, and nobody has modified them since. In some cases, we want to enforce confidentiality as well as integrity on these resources, always within the scope of an SFTC.

As mentioned before, SmartFrog can use web servers to dynamically load resources. Web servers are not part of the SFTC, although they host resources that are trusted by members of the SFTC.

SmartFrog uses PKI (Public Key Infrastructure) based on X509 certificates to provide principals of the SFTC with credentials that justify they are valid members of the community.

Java 2 security mechanisms are leveraged to create a SFTC. In particular, the java security policy is set so that, when enforced by a SecurityManager, gives full privileges to classes loaded from signed jars (using a trusted key for the SFTC), and none otherwise. These mechanisms are extended to other resources apart from classes, such as configuration descriptions, so that they are only loaded if they are in a signed jar (same signing key as before). This is particularly useful when the jars are hosted by web servers that are not part of the SFTC.

RMI calls are tunneled over SSL using the JSSE API and Sun's reference implementation. SmartFrog forces mutual authentication in the SSL sessions based on the 1024 bits RSA public/private keys that are discussed above. Only the SFTC CA keys are part of our trust assumptions so, by validating the other partner's X509 certificate chain, SmartFrog knows that it belongs to the community. Current SSL session settings include triple DES encryption, with HMAC SHA-1 for message authentication. In addition, a similar mechanism is used to protect access to the RMI registry.

SmartFrog supports dynamic loading of stubs during RMI calls, or arbitrary resources in signed jar files using an enhanced RMI class loader. In both cases the loading sources is restricted to a configured codebase, and use the Java 2 mechanisms for signed jar files described above.

All SmartFrog core classes use the security hooks described above for loading resources and communicate with other peers. In addition, the set-up of the security mechanisms is done as early as possible in the initialization of daemons to minimize exposure.

With the present security model special care should be taken before authorizing that a particular application can be deployed in the platform. Deployed components share the same environment and privileges of the SmartFrog infrastructure, and therefore if security is compromised in one of them, it could potentially affect the rest. Moreover, if one member of the trusted community is compromised, the whole community is at risk. We are currently working on extensions to the security model to reduce these risks.

Additional Tools

Additional tools are available to complement the SmartFrog core framework.

A development tool for easing the creation, validation and inspection of configuration descriptions is available. The development tool can also deploy and manage SmartFrog applications.

Java Management Interfaces (JMX)[4][5] has been fully integrated with the SmartFrog core[3].With JMX a SmartFrog Application can be managed and accessed using standard management interfaces. Also, a complete JMX infrastructure can be fully deployed, configured and distributed using SmartFrog.

BeanShell [6] Java interpreter is integrated with the SmartFrog language offering a lightweight of embedding component behavior directly into SmartFrog configuration data.

Service Location Protocol (SLP) [7][8] has been added as additional service to the core framework providing an alternative discovery method.

Experiences using the framework

The concepts and technology in SmartFrog have already been deployed in production systems. SmartFrog is also been used to deploy and configure Grid applications [2] using the Globus tool-kit. This service is to be presented in Global Grid Forum 8.

Besides, SmartFrog is also the system used to deploy and manage demonstrators of *adaptive solutions* in the HP's Utility Data Centre [9]. In the deployment of these adaptive solutions we automatically configure and ignite not just the infrastructure, but the applications that run on it (such as Apache), via unified descriptions of the infrastructure and services that are required.

Future Work

SmartFrog continues evolving as the main demonstrator of our research program. Some of the improvements for the future are: a new description language with support for predicates used to check the correct application of configurations, a new security model with multiple domains of trust and with a finer degree of control and a novel reliability service for the framework and for the applications deployed using the framework. Also, the set of components offered will be extended with basic components offering different failure-recovery modes, lifecycle or functionality.

Conclusion

Together, the SmartFrog language, component model and the deployment infrastructure allow for great flexibility in system configuration. It is possible capture many system-wide properties at the description level, resulting in very different runtime system configurations, without the software components needing to be aware of this, or implemented differently.

Finally, SmartFrog is a framework, not a specific application. This means there is great flexibility in how it can be used. The current version of the system has a language system, plus a minimal core framework. The core framework is populated with useful components, such as those that sequence other components, or those that support reliability 'patterns', but these are optional and can all be replaced with components that are more suitable for a particular context.

Acknowledgements

The authors would like to thank Marc Nijdam for his important role in the development of SmartFrog.

References

[1] Paul Anderson, George Beckett, Kostas Kavoussanakis, Guillaume Mecheneau and Peter Toft. Experiences and Challenges of Large-Scale System Configuration. March 2003. http://www.epcc.ed.ac.uk/gridweaver/WP2/report2.pdf.

[2] Grid Weaver Project. eScience Grid Core Programme. http://www.gridweaver.org

[3] Julio Guijarro, Manuel Monjo. Framework for managing large scale component-based distributed applications using JMX. Presented at HPOVUA 2002. May 2002.

[4] Sun Microsystems. *Java Management Extensions Instrumentation and Agent Specification, v1.0*. July 2000, Sun Microsystems. http://java.sun.com/jmx.

[5] Juha Lindfors, Marc Fleury. *JMX. Managing J2EE with Java Management Extensions*. 2002 Sams Publishing.

[6] BeanShell. http://www.beanshell.org.

[7] James Kempf, Pete St. Pierre. Service Location Protocol for Enterprise Networks. 1999 Wiley Computer Publishing.

[8] SLP information: http://www.openslp.org/.

[9] Scarlet Pruitt. HP dabbles in utility data center services. IDG News Service. May 22, 2003. http://www.idg.net/ic_1317170_9677_1-5044.html