A SmartFrog Tutorial

Colin Low

Hewlett Packard Laboratories
Filton Road,
Bristol BS34 8QZ
colin.low@hp.com

Version 1.2 rev. 3

This document provides an introduction to SmartFrog, and provides a step-by-step guide to system configuration and deployment using the framework. It brings together material from *The SmartFrog Reference Manual*, *The SmartFrog User Guide* and *The SmartFrog API (Javadoc)*, which should be consulted for details. Sources and tutorial available from www.smartfrog.org.

1 Introduction

SmartFrog is a tool. With any tool, it is important to know how it will make your life simpler, when to use it, and what it does well. What follows is an example that illustrates some of the advantages of using SmartFrog.

Let us suppose you are developing an application with several autonomous components. These components could be parts of a distributed application, with components distributed across multiple hosts. It is necessary to test the individual components, and you realise that to test **component1** you will need to have each of **component2** to **component7** in place. Whenever you run a test, you will have to activate seven components with all the correct configuration parameters. You may have to carry this out dozens of times until testing is complete.

You could use a terminal window to activate seven processes manually each time you run a test. But what happens when something goes wrong and your component fails to execute correctly? The six auxiliary processes will need to be killed and restarted.

Many developers will have developed solutions to this kind of problem using shell scripts. This is acceptable for simple situations, but as tests become more demanding, and you begin to deploy components over several hosts, the whole business of deployment and lifecycle management becomes extremely time consuming and error-prone. With interruptions and tiredness, it is easy to neglect one of the steps and waste time trying to figure out what has gone wrong. Wouldn't it be good to have a tool that enables you to deploy complex applications over hundreds of computers? This is an example of the many things that SmartFrog can do.

I have been involved in the development of several large distributed applications, and what I realised when using SmartFrog for the first time was that even if I had used it only during the test and integration phase, it would still have been worth learning because of the huge amount of time it saved in setting up test configurations. The building block approach meant I could create test configurations as easily as plugging together Lego. This paid off when the application went live, because I had been able to spend much more time testing.

It paid off in other ways too. When my co-workers wanted to integrate their components with mine, I gave them my configurations and they slotted in their components. We were sharing code through a code repository, so all they needed were my SmartFrog configurations to create test systems that included separately developed components. As we moved from the test environment to the live environment, the majority of the basic configurations remained unchanged; what changed was the enclosing configuration in which the building blocks were composed into the final application. What we built was an extremely complex distributed system providing a commercial application deployed dynamically over hundreds of computers. I doubt that it would have been possible to create it in the time available without SmartFrog.

Although SmartFrog is conceptually simple, it would be wrong to suggest it is simple to learn. This is not because it is arcane … it is more a case that its flexibility provides many alternative ways to achieve an end, and the novice is likely to be overwhelmed.

The purpose behind this tutorial is to provide the fragments that explain how to achieve the most common things you are likely to want to do with SmartFrog. I was motivated to do this because of my own experiences with learning and using

SmartFrog. I am not on the SmartFrog team; on the other hand, I have direct access to the authors and implementers of SmartFrog, so I hope I have been able to keep the errors of understanding and implementation to a minimum.

2 SmartFrog Basics

The capabilities that SmartFrog provides are:

- **Configuration:** the ability to describe the autonomous components of an application and to compose these components into a distributed application in a single description.
- **Deployment:** the ability to deploy a configuration over a set of computing resources.
- **Lifecycle:** the ability to ensure that components progress through their lifecycles – deploy, start and terminate - in an orchestrated way.
- **Discovery and Communication:** components can locate other components in an application and communicate with them, both in static placements, and dynamically at run time..

3 Programming Basics

SmartFrog is written in Java. It assumes that your components are written in Java. You will require experience with Java.

The SmartFrog system expects that the environment variable SFHOME is set to the directory containing the SmartFrog distribution (see comment). Your PATH variable should include `SFHOME/bin` so that you can run SmartFrog scripts.

Comment: the SmartFrog source distribution, when built, contains a folder `dist`. In this folder are directories `bin`, `lib` etc. I am assuming that SFHOME points to this folder.

SmartFrog needs to find all the classes used by your application components. The best way to achieve this initially is to set the environment variable SFUSERHOME to point to a directory containing the Java Archive (JAR) files for your classes.

Each host on which you intend to load and run components will require a SmartFrog daemon process. The daemon process will access the JAR files in SFUSERHOME. If you alter your code and recompile to produce a new JAR file, the daemon will need to be restarted. Failure to do this is a common error.

The SmartFrog distribution contains a script in `SFHOME/bin` called **sfDaemon**. This script, which has Windows and UNIX variants, will start the SmartFrog daemon process on a host. If you intend to run SmartFrog on many hosts, it is useful to create your own script capable of starting and stopping the daemon on a collection of hosts. There are some predefined SmartFrog templates that help in doing this[1]. Another way to achieve this would be to use a tool such as *Cluster SSH* (at clusterssh.sf.net).

If you intend to modify your application code frequently (e.g. because you are testing) then some way to propagate the code to each host will be required. A good method is to place your JAR files on a fileserver, and for each host to mount the remote fileserver directory as the local path pointed to by SFUSERHOME. *Each time you change your code* on the server, each host will see your changes. It is at this point that

you will need to restart the SmartFrog daemon process on each host. It is possible to use an HTTP server to serve classes to SmartFrog and avoid this restart – this is described in the *SmartFrog Reference Manual*.

A beginner's configuration is to have your Java classes on your local host, and to start the **sfDaemon** script manually from a command-line shell.

4 Creating a SmartFrog Component

The simplest SmartFrog component is a class that extends the SmartFrog class **PrimImpl**. Your component should also implement the **Prim** interface so that the Java RMI compiler (rmic) functions correctly.

Your component will inherit many behaviours through **PrimImpl**. In particular, your component will participate in SmartFrog lifecycle management via calls to **sfDeploy()**, **sfStart()** , and **sfTerminateWith(…)**. If you fail to override these methods then your component will deploy, start and terminate at the correct times, but none of your code will be called. It is normal to override these three methods, so that your component initialises itself in **sfDeploy()**, it commences active operation in **sfStart()**, and closes down in **sfTerminateWith(…)**.

A template class **MyPrim** that overrides these methods is shown below in Example 1.

```
package org.smartfrog.examples.tutorial;

import org.smartfrog.sfcore.prim.Prim;
import org.smartfrog.sfcore.prim.PrimImpl;
import org.smartfrog.sfcore.prim.TerminationRecord;
import org.smartfrog.sfcore.common.SmartFrogException;
import org.smartfrog.sfcore.common.SmartFrogResolutionException;

import java.rmi.RemoteException;

public class MyPrim extends PrimImpl implements Prim {
  /* any component specific declarations */
  public MyPrim() throws RemoteException {
  }

  public synchronized void sfDeploy() throws
RemoteException,SmartFrogException {
    super.sfDeploy();
    /* any component specific initialization code */
    sfLog().out("SmartFrog "+sfCompleteName()+" deployed");
  }

  public synchronized void sfStart() throws
RemoteException,SmartFrogException {
```

```
    super.sfStart();
    /* any component specific start-up code */
    sfLog().out("SmartFrog "+sfCompleteName()+" started");
  }

  public synchronized void sfTerminateWith(TerminationRecord tr) {
    sfLog().out("SmartFrog "+sfCompleteNameSafe()+" terminating");
    /* any component specific termination code */
    super.sfTerminateWith(tr);
  }
  /* any component specific methods go here*/
}
```

Example 1. MyPrim.java

If **MyPrim** is started concurrently with several other components using the default **Compound** behaviour (see below), then all components will progress through their lifecycle in an orderly and synchronised way. No component will enter **sfStart** until all components have completed **sfDeploy**.

If you have tried to achieve multi-component deployments using ad-hoc methods, you will appreciate how many problems can be caused by race conditions during initialisation. A widely experienced problem is that **component2** comes ready before **component1**, and **component2** reports errors because **component1** does not appear to be present. If **component2** reports a failure and quits, the problem is compounded because **component1** is left in isolation, and may require manual tidying-up. This kind of initialisation can be implemented using arbitrary wait or retry loops, which may work on one particular configuration of equipment, and break immediately the configuration changes.

SmartFrog avoids these problems. If one component experiences an error during initialisation and terminates, all components will be terminated cleanly. The precise mechanics as to how this happens is described in section 8, which describes the SmartFrog lifecycle model.

Once you have created **MyPrim**, you should describe it using a SmartFrog component description, which you could save as `MyPrim.sf`.

```
#include “org/smartfrog/components.sf”
MyPrim extends Prim {
   sfClass "org.smartfrog.examples.tutorial.MyPrim";
}
```

Example 2. MyPrim.sf

The description in Example 2 tells SmartFrog that there is a **Prim** component, and the code for it can be found in a class called `MyPrim.class`.

Note Well: in general, SmartFrog components are remote objects in the sense understood by Java Remote Method Invocation (RMI). If you haven't used RMI, or aren't familiar with the idea of calling methods on remote objects, it would be good to familiarise yourself with the concepts. In particular, you will need to know how to compile the **MyPrim** class using the Java `rmic` compiler. If you fail to do this, SmartFrog will fail at runtime when you attempt to start your component. The failure message is not particularly edifying, so do not neglect this point. In the specific case where a component has no remote methods, then `rmic` compilation can be avoided by adding the attribute "sfExport false;" to the description. This has been included in Example 3, but commented out.

Comment:

SmartFrog is built using ANT and the build.xml script in the top-level SmartFrog directory. This is a good, if somewhat complex, example of how to build SmartFrog components.

You can try out your version of **MyPrim** by creating a new configuration description.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/tutorial/MyPrim.sf"

sfConfig extends Compound {
  MyPrimTest extends MyPrim {
    // uncomment sfExport if the component has no RMI interface
    // sfExport false;
  }
}
```

Example 3. MyPrimTest.sf

The line beginning **sfConfig** is equivalent to the "main" entry point in Java: it tells SmartFrog where to begin with its deployments. In this case there is only a single component to deploy, the component **MyPrimTest**. This in turn is an extension of **MyPrim**, which was declared previously, and SmartFrog can locate the code described in Example 1 via the **sfClass** attribute.

Why the mystery of the two declarations, first **MyPrim**, then **MyPrimTest**? The reason is that you may wish to deploy **MyPrim** in many different configurations. The **MyPrim** declaration allows you to specify the defaults for the configuration of this single component in isolation from any other components. The **MyPrimTest** declaration can override any of these defaults. In this example there is nothing to override, but the concept will become clearer in later sections. You are not obliged to follow this convention, but you may find that if you have to create many different configurations using the same component (during testing for example), then having a default declaration is a useful level of abstraction.

Comment:

sfConfig can extend any component derived from **Prim**. If the application contains multiple components it is common for **sfConfig** to extend **Compound**, and you will see many examples of this. However, it could have extended **MyPrim**, as.there is only the single component in this application.

5 Running a SmartFrog Component

You can start your component using the script **sfStart**, found in SFHOME/bin, as in Example 4:

```
sfStart localhost TEST1 org/smartfrog/examples/tutorial/MyPrimTest.sf
```

Example 4

Here the first argument is the hostname of the host (SmartFrog root process) where you want to initiate deployment, the second argument is the name you want to give to the component, and the third argument is the path to the script. I have assumed the script is in a JAR file located in the folder pointed to by `SFUSERHOME`. You will need to start **sfDaemon** on localhost prior to running **sfStart**, or the **sfStart** will fail. So long as the daemon is running[1], you can use **sfStart** many times, but you cannot use the component name `TEST1` again if `TEST1` is still running.

You can terminate the component as in Example 5:

Comment:

You can compile your new component manually using javac and rmic or you can create a directory called "tutorial" under dist/src/org/smartfrog/examples and place there MyPrim.java MyPrim.sf and MyPrimTest.sf. The ant build file included in dist automatically compiles and packages inside sfExamples.jar any new files added under src/org/smartfrog/examples. To run rmic for MyPrim.class and create its rmi stubs we need to edit the file dist/rmitargets and add "org/smartfrog/examples/tutorial/MyPrimclass".

Now you only need to go to dist/ and run "ant build" to get your new component ready to be used.

```
sfTerminate localhost TEST1
```

Example 5

Alternatively you can kill the SmartFrog daemon. The orderly way to achieve this is

```
sfStopDaemon localhost
```

Example 6

Terminating the shell from which the daemon was started will also do the trick. Control+C will attempt an orderly termination, and if that fails, a second Control+C will kill the daemon.

[1] To check if a daemon is running in a node you can use: "sfPing hostname"

6 Component Configuration

Many programs are dependent in some way on their external environment, and configure themselves at runtime using values that are often read from a file, such as the `MyProgram.ini` files that abound in Windows.

This becomes much harder to achieve in a large distributed application containing many components. Each component may have its own configuration data, but there may also be dependencies between values used by different components, and values that are shared by many components. Even though the components may be distributed, the global configuration data is shared, and it may be sufficiently complex and structured to require navigation tools. This is what SmartFrog provides in its structured data syntax.

SmartFrog overloads its syntax in several ways, and this can make its meaning somewhat opaque to the beginner. The most important way in which SmartFrog overloads its syntax is that it mixes configuration data with deployment information. In other words some of the declarations are used to pass data to (and between) components, while other declarations *that look superficially similar* result in component deployments. It is common for declarations to achieve both ends simultaneously.

Take the previous component declaration `MyPrim.sf`. This has been extended by the addition of three new values: **debug**, **retryCount**, and **databaseRef,** as in Example 7**.**

```
#include “org/smartfrog/components.sf”

MyPrim extends Prim {
   sfClass "org.smartfrog.examples.tutorial.MyPrim";
   debug true;
   retryCount 10;
   databaseRef TBD;
}
```

Example 7

The purpose of **debug** is to switch off and on debugging statements within the component. The **retryCount** is the number of times the component should attempt an action before giving up. The **databaseRef** is a reference to a database. Its value is given as `TBD` – “to be defined”; that is, the value needs to be defined, but isn’t defined at this point. These are common examples of values you may wish to change, but which are normally stable over long periods of time.

The **MyPrim** declaration now achieves two ends: it defines a **Prim** component, including a specification of the class implementing the component, and it supplies configuration values to the component. We can extend the **MyPrimTest** script in Example 3 to supply the missing TBD database reference.

```
#include “org/smartfrog/components.sf”
#include “org/smartfrog/examples/tutorial/MyPrim.sf”

sfConfig extends Compound {
  MyPrimTest extends MyPrim {
    databaseRef “a reference to a DB”;
  }
}
```

Example 8

Note that any of the values defined in **MyPrim** can be modified within **MyPrimTest**, but in this case the only one modified is the undefined value **databaseRef**.

The values in a component declaration are accessible within the implementation of a component inheriting from **PrimImpl** using the **sfResolve** method call.

```
void getConfig(){
  try {
    Boolean debug = (Boolean)sfResolve(“debug”);
    Integer retryCount = (Integer)sfResolve(“retryCount”);
    String databaseRef = (String)sfResolve(“databaseRef”);
  } catch(ClassCastException ccex){
    ...
  } catch(SmartFrogResolutionException srex){
    ...
  } catch(RemoteException rex){
    ...
  }
}
```

There are many variants of the **sfResolve** methods; the **PrimImpl** class inherits from **RemoteReferenceResolverHelperImpl**, and you will find the many variant forms described in the JavaDoc for this class. The form used above is one of the simplest; it

Example 9

takes a **ReferencePart** string, and attempts to resolve it in the configuration context of the component. Put more simply, the **MyPrim** class has access to the contents of the **MyPrimTest** declaration in Example 8, and **sfResolve(…)** looks for an attribute string (e.g. “debug”) specified in that declaration.

These values are resolved at runtime, and so certain kinds of error will only be detected at runtime. There are three kinds of exception you should be aware of.

Firstly, this version of **sfResolve** returns an **Object**, and this should be cast to the correct class of Java object. The SmartFrog parser can deduce the kind of object from

the syntax. For example, SmartFrog parses “debug” as a **Boolean** because it has the value **true**, and you should cast it to a **Boolean**. Failure to do this correctly will cause a **ClassCastException**.

Secondly, SmartFrog may not be able to resolve the reference, and produce a **SmartFrogResolutionException**.

Lastly, SmartFrog may need to use remote method calls to resolve certain kinds of reference (e.g. LAZY references, or a reference to the context of another component), and in turn this could cause a **RemoteException**. This could happen if the underlying communication infrastructure is causing problems.

7 Deployment

The code and Smartfrog configuration snapshots provided so far provided almost everything you need to create a SmartFrog component, supply it with configuration parameters, deploy it, and then access the configuration parameters from within the running component. These are the basics; if you understand them you can do a great deal with SmartFrog. However, most of the power of SmartFrog comes from deploying multiple components. There is only a small amount of additional

```
#include “org/smartfrog/components.sf”
#include “org/smartfrog/examples/tutorial/MyPrim.sf”

sfConfig extends Compound {
  MyPrimTest extends MyPrim {
    databaseRef “a reference to a DB”;
  }
}
```

Example 10

complexity involved in understanding how to do this. In fact, we have already done it, albeit with a single component, as shown in Example 10.

The reserved name **sfConfig** is defined to extend **Compound**. A **Compound** is a component capable of monitoring the state of a collection of children, and the children in turn can be **Compound** or **Prim** components. The simplest structure for SmartFrog components is a tree, with **Compound** components as nodes, and **Prim** components as leaves.

A more complex deployment is shown in Example 11.

This example deploys three components: an enclosing **Compound**, and two **Prim** components. One **Prim** component is a server, the other component a client of the server. Both components are deployed onto specific hosts using the **sfProcessHost** attribute, an attribute that is interpreted by SmartFrog as a placement directive. The server is placed on host “server.hp.com”, the application on “localhost”.

It is normal for a client application to need to know the internet address and port number of a server. The server also needs to know what port number to use. This achieved by defining both values at the top level of the configuration, and using *references* (see Section 11) in both component definitions that refer to the PARENT environment. It is now impossible to deploy the client and server without the client and server agreeing on location and port.

In this example both client and server share the same lifecycle. If the client terminates, so does the server. This is because both components are children of an enclosing **Compound**. A shared lifecycle is not realistic for a live client-server application (and there are simple ways to change this behaviour), but it is extremely convenient for testing – when you kill the locally deployed client, the remotely deployed server is automatically removed. This kind of script enables many cycles of testing without the nuisance of having to tidy-up remote components manually. When the time comes to detach the lifecycle of the server from the client, this is easily achieved.

Comment:

There is a variant of **Compound** called **DetachingCompound**, with some interesting behaviours. The reader is encouraged to look at the comments in the *DetachingCompound.sf* prototype under the SmartFrog source tree.

In addition, any **Prim**-derived component can detach itself from its parent by calling **sfDetach().**

```
// File ClientApp.sf
#include "org/smartfrog/examples/tutorial/MyPrim.sf"
ClientApp extends MyPrim {
}
```

```
// File ServerApp.sf
#include "org/smartfrog/examples/tutorial/MyPrim.sf"
ServerApp extends MyPrim {
}
```

```
// File ClientServerTest.sf

#include "org/smartfrog/components.sf"
#include "org/smartfrog/examples/tutorial/ClientApp.sf"
#include "org/smartfrog/examples/tutorial/ServerApp.sf"

sfConfig extends Compound {
    serverHost "server.hp.com";
    serverPort 4242;

    MyClient extends ClientApp {
        sfProcessHost "localhost";
        server PARENT:serverHost;
        port   PARENT:serverPort;
    }

    MyServer extends ServerApp {
        sfProcessHost PARENT:serverHost;
        port          PARENT:serverPort;
    }
}
```

Example 11. ClientServerTest.sf

Before going into the esoteric details of lifecycle management, it is useful to look at the default component lifecycle in more detail.

8 Lifecycle

8.1 Transitions

Components in a distributed system often have dependencies. If **component1** needs to communicate with **component2**, and the initialisation of **component2** has been delayed, then **component1** may report an error. **Component1** may then fail in turn, causing a cascade of failures. If this kind of race condition is intermittent, it can be extremely time-consuming to diagnose, and remedying the problem may require *ad-hoc* protocols which are likely to cause further problems (as bolt-on, *ad hoc* protocols so often do ….)

SmartFrog provides internal protocols for ensuring that a collection of components move through their lifecycles in a synchronised and consistent way. The normal states and transitions for a SmartFrog component are (see Figure 1: Component Lifecycle):

- **Constructed:** the default constructor has been called. This state terminates when the **sfDeployWith(ComponentDescription)** method is called.

- **Instantiated**: the component has been created, but application-specific initialisation has not taken place. This state terminates when **sfDeploy()** has been called. Any component derived from **PrimImpl** is expected to override **sfDeploy()** to provide application specific initialisation.

- **Initialised:** the component is initialised, but not yet started. This state terminates when **sfStart()** has been called. Any component derived from **PrimImpl** is expected to override **sfStart()** to provide application specific processing.

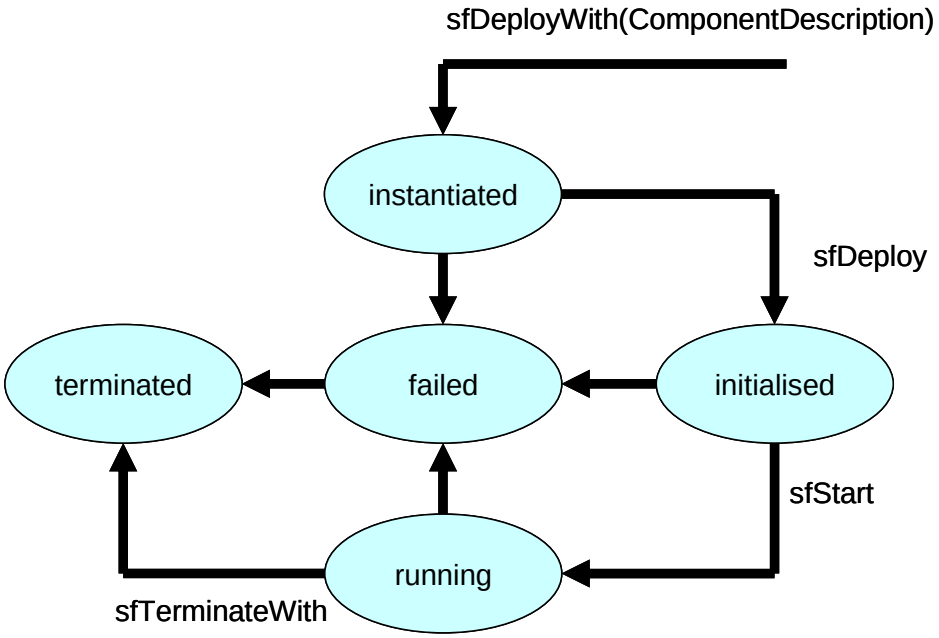

Figure 1: Component Lifecycle

- **Running**: **sfStart()** has been called, and the application code is now running.
- **Terminated**: there are many ways in which a component can enter the **Terminated** state. It can be triggered by an up-call from a child component, a down-call from a parent component, or an internal call to **sfTerminate(…)**. This is discussed in more detail in the following section.

The simplest way to create a collection of components with a linked lifecycle is to make them children of a **Compound** component. The semantics of **Compound** are that all children will be instantiated before any are initialised. All children will be initialised before any are running. If any child terminates, all will be terminated.

An example of this can be seen in Example 12, in which four components are created. The first is a parent **Compound** named **sfConfig**, and the other three are **Prim** children named **MyPrimTest1**, **MyPrimTest2**, and **MyPrimTest3**.

```
#include “org/smartfrog/components.sf”
#include “org/smartfrog/examples/tutorial/MyPrim.sf”

sfConfig extends Compound {
  MyPrimTest1 extends MyPrim {
    databaseRef “a reference to a DB”;
  }
  MyPrimTest2 extends MyPrim {
    databaseRef “a reference to a DB”;
  }
  MyPrimTest3 extends MyPrim {
    databaseRef “a reference to a DB”;
  }
}
```

Example 12

What SmartFrog guarantees is that if any of these components have entered **sfStart()**, then *all* of the components will have completed and exited **sfDeploy()**.

It sometimes happens that **sfDeploy()** fails in one of the components because an external resource is not available. If there is insufficient internal logging in your application code then the cause may not be obvious. The obvious external symptom is that no SmartFrog component has started. It looks as if everything has failed. Indeed, it is probable that if the failure has resulted in an **Exception**, every component will have terminated.

It is easy to blame SmartFrog when this happens. From superficial appearances, it looks as if SmartFrog has failed to deploy the application. The answer is to ensure that there is sufficient diagnostic output in each component so that any exceptional condition during **sfDeploy()** is clearly reported.

It is also important to remember that if any component terminates, all will terminate, and the internal call to **sfTerminateWith(…)** will occur concurrently with normal

operation. For example, this might mean closing a database connection in the middle of a database operation, which will cause more exceptions. One can lose a considerable amount of time chasing down a problem inside the database code, when the real problem is that some other SmartFrog component failed, and triggered the termination of the component carrying out the database action.

The answer to this is to study the diagnostic output carefully, and not leap to incorrect conclusions based on a small fragment of diagnostics from one component. The default behaviour is that if any component in a **Compound** terminates, all components will terminate, and you must be aware that termination will be a thread running concurrently with your main processing thread. Your main processing thread should be written in such a way that the thread resulting from a call to **sfTerminateWith** can tell it to shutdown cleanly – input/output needs to terminate cleanly, and any **Exceptions** should be handled cleanly.

8.2 Termination Sequence

The termination sequence is described with reference to a tree of **Compound** and **Prim** components, the **Compound** components being nodes in the tree, and the **Prim** components being leaves. This is shown in Figure 2 below.

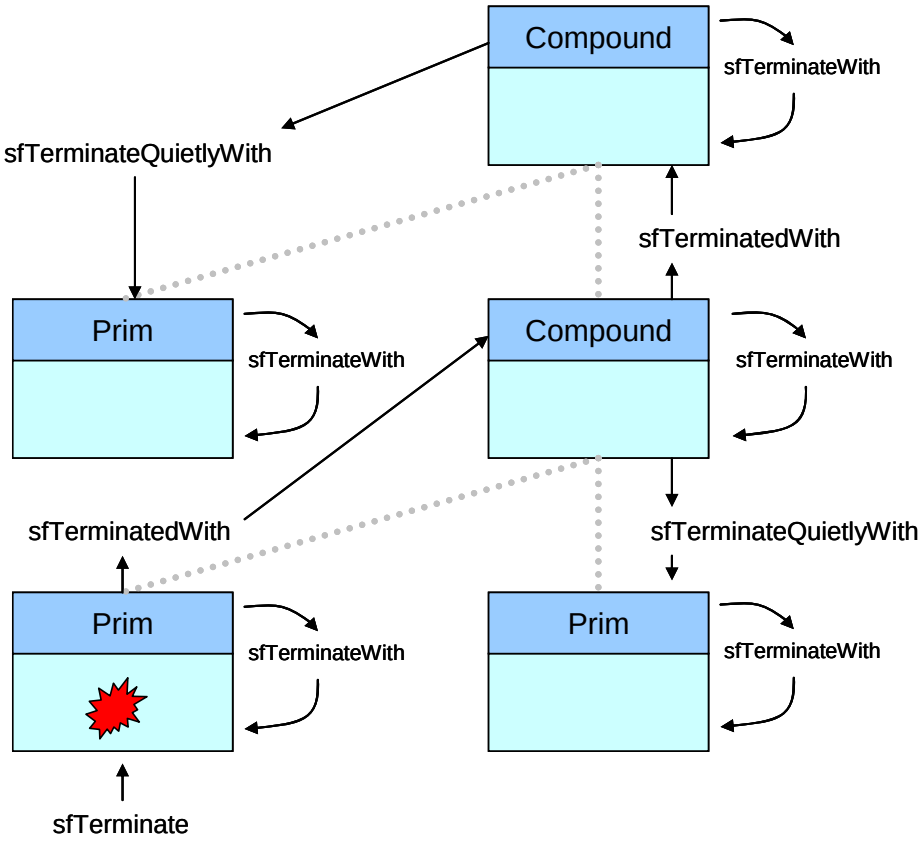

Figure 2: Termination Sequence

The bottom-left **Prim** component terminates, and this is marked by a "bang!" symbol. There are varies reasons why it might be terminated, but for the purposes of

presentation we can assume that its **sfTerminate** method has been called. This component could have encountered an error and called **sfTerminate** internally, or it could have been called externally.

The terminating **Prim** calls its parent with **sfTerminatedWith(…)**, passing a **TerminationRecord** containing details of the termination. This process is repeated up the tree, with each child calling **sfTerminatedWith(…)** until the root of the tree is reached. Each parent then communicates down the tree to its children using **sfTerminateQuietlyWith(…)**. In this way a failure in any part of the tree is communicated so that all components are terminated.

Each component, whether **Prim** or **Compound**, then executes **sfTerminateWith(…)**. It is normal to override this method with application-specific termination code – see the discussion in the previous section..

8.3 TerminationRecord

A call to **sfTerminate()** requires the construction of a **TerminationRecord**. It may not be obvious how to do this, as one of the parameters is a **Reference** to the failed component.

If a component is calling **sfTerminate()** on itself, then it is valid to leave this parameter as **null**, as the **Prim** parent class will fill in the **Reference** if it is possible.

```
TerminationRecord tr = TerminationRecord.abnormal("some kind of
failure message", null);
this.sfTerminate(tr);
```

Example 13

If you need to fail a component because of a problem in another component, then the code in Example 14 might be more appropriate:

```
Reference failedName = null;
Prim failed = ... ; // a failed Prim component

try {
   failedName = ((Prim) failed).sfCompleteName();
} catch (Exception ex) {
   // ignore, leave null
}

TerminationRecord tr = TerminationRecord.abnormal("some kind of
failure message", failedName);
this.sfTerminate(tr);
```

Example 14

8.4 Failure

A SmartFrog application can fail spontaneously because individual components lose contact with each other. This could happen because of a communication failure in the underlying network, or a host failure that takes out a component. There is a default liveness mechanism based on ping messages between parents and children that will detect failures and trigger the default termination process described in the previous section.

A SmartFrog **Compound** component tests the liveness of its children using the **sfPing()** method. **sfPing()** is also called internally by each **Prim** component. An external call from a parent resets a count; a timer-driven internal call decrements the same count. When the count reaches zero, **sfLivenessFailure()** is called.

The default behaviour of **sfLivenessFailure()** in **PrimImpl** is to call **sfTerminate** and so initiate the termination sequence described in the previous section. In this way a host or communication failure will result in all application components terminating cleanly even when a component is isolated from other components.

There are two configurable attributes that can be used to influence this behaviour:

- **sfLivenessDelay**: defines how often to call **sfPing()** (in seconds). Note that this controls the rate of **sfPing**s to external components *and* the rate at which the component counts down **sfLivenessFactor** to zero and then declares liveness failure.
- **sfLivenessFactor**: how many multiples of **sfLivenessDelay** to wait before declaring a failure.

The failure-detection and termination mechanisms in SmartFrog can be overridden to create a measure of fault tolerance. A fault-tolerant application may spawn new components when a failure is detected. The basic functionality for doing this can be found in **CompoundImpl**. There are four basic steps:

- Detect that a component has failed
- Identify the failed component
- Identify a suitable host
- Deploy a new instance of the failed component on the new host.

CompoundImpl inherits from **Prim**, so it possesses all the behaviour of a **Prim** component. In addition, it implements the **ChildMinder** interface. These methods can be used to add and remove children explicitly. **CompoundImpl** also implements the interface **Compound**, which has methods for deploying new components using a parsed **ComponentDescription** – this can be extracted from the **Context** of the **CompoundImpl** itself using the usual **sfResolve** methods already used in section 6.

The main task is to reimplement the **sfLivenessFailure()** method. Instead of a child failure terminating the parent **Compound**, one can trap the failure, remove the child, deploy a new component as a replacement, and add the new component as a child.

9 Configuration Revisited

One of the main purposes of a SmartFrog description is to describe a set of components to be deployed. The default component for managing a collection of

components is the **Compound**. A **Compound** can contain a **Compound** as well as a **Prim**, which leads to an arbitrarily deeply nested description which has the structure of a tree. This is shown in Example 15.

```
sfConfig extends Compound {
   desc1 extends Prim{
   }
   desc2 extends Compound {
      desc3 extends Prim {
      }
      desc4 extends Prim {
      }
   }
}
```

Example 15

When this description is parsed, it is parsed into objects implementing the **ComponentDescription** interface; in practice these are **ComponentDescriptionImpl** objects. This has the structure of a tree. Although you may never need to be aware of this, I found that understanding it greatly helped me in visualising what SmartFrog was doing under the hood.

9.1 Component Descriptions

A **ComponentDescriptionImpl** object is the internal, parsed version of a textual component description such as **desc3** above. It contains the set of attribute/value pairs described by the component, and linkage to parent and children

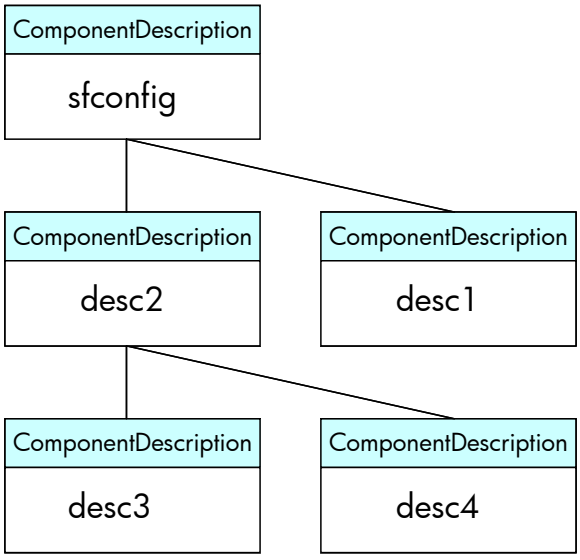

Figure 3: A ComponentDescription Hierarchy

ComponentDescriptionImpl objects**.**

When the description in Example 15 is parsed, it is turned into a tree of **ComponentDescriptionImpl** objects as shown in Figure 3. A **ComponentDescriptionImpl** is something that can be deployed as a SmartFrog component by the SmartFrog deployer, if it meets certain well-formedness criteria (e.g the **sfClass** attribute is defined).

The attribute/value pairs in each **ComponentDescriptionImpl** object are actually stored in an associated **ContextImpl** object, which can be thought of as a **java.util.Hashtable** with ordering properties – in fact, it is derived from **java.util.Hashtable**. The **ComponentDescriptionImpl** child objects underneath a given **ComponentDescriptionImpl** object are values in its associated **ContextImpl**. For example, the object corresponding to **desc2** in Example 15 above will contain "desc3" as a key in its **ContextImpl**, and the value will be a **ComponentDescriptionImpl** object containing the attributes defined in **desc3** (there aren't any).

When a **ComponentDescriptionImpl** is deployed, an object derived from **PrimImpl** is created, and the **ContextImpl** that describes the attribute/value pairs is copied and attached to the **PrimImpl.** In other words, the tree of component descriptions is transformed into a tree of deployed components. However, any component descriptions tagged as LAZY are not transformed into **Prim** components, and remain in the Context as **ComponentDescriptionImpl** objects.

This may seem somewhat arcane. For most purposes it is not necessary to understand **ComponentDescriptionImpl** objects in any depth. The primary reason for introducing them is that they are the parsed, internal descriptions of components that may be deployed, and an application that deploys new components dynamically in response to changing conditions will almost certainly make use of these descriptions.

10 Dynamic Deployment

The following example demonstrates how to deploy a component at runtime from within another component. The SmartFrog configuration for this is shown in Example 16 below.

Two components are defined as part of the Java package “recipes”. The first component is **Parent**, and **Parent** is a **Compound** that contains a TBD reference to a

```
// File ParentChildExample.sf
#include "org/smartfrog/components.sf"

Parent extends Compound {
   // a parent component capable of spawning a child
   sfClass "org.smartfrog.examples.tutorial.recipes.Parent";
   myChild TBD;
}

Child extends Prim {
   // a potential child component
   sfClass "org.smartfrog.examples.tutorial.recipes.Child";
}

sfConfig extends Compound {
   myParent extends Parent {
      myChild extends LAZY Child; // resolve the child reference here
   }
}
```

Example 16. ParentChildExample.sf

child component, and an **sfClass** attribute that defines the Java class. The **Child** component is a **Prim** component that contains only the **sfClass** attribute.

The deployed configuration, defined within **sfConfig**, consists only of the **Parent** component. The TBD **myChild** attribute is now defined as the previously declared **Child** component, but this is not deployed because the reference to it is defined as LAZY. This means that it is parsed, and the prototype reference is resolved, but it is not deployed.

```
// Parent.java
package org.smartfrog.examples.tutorial.recipes;

import java.rmi.*;
import org.smartfrog.sfcore.common.*;
import org.smartfrog.sfcore.prim.*;
```

```
import org.smartfrog.sfcore.compound.*;
import org.smartfrog.sfcore.componentdescription.*;

public class Parent extends CompoundImpl implements Compound {

  ComponentDescriptionImpl child = null;

  public Parent() throws RemoteException {
  }

  public void sfDeploy() throws RemoteException, SmartFrogException {
        super.sfDeploy();
        try {
            sfLog().out("SmartFrog "+sfCompleteName()+" deploying");
            child = (ComponentDescriptionImpl)sfResolve("myChild");
            sfLog().out("Child configuration reference found");
            sfLog().out("SmartFrog parent process deployed");
        } catch (Throwable t) {
            t.printStackTrace();
            throw SmartFrogException.forward(t);
        }
  }

  public void sfStart() throws RemoteException, SmartFrogException {
        try {
            super.sfStart();
            sfLog().out("SmartFrog parent process starting");
            sfCreateNewChild("Child", child, null);
            sfLog().out("SmartFrog parent process started");
        } catch (Throwable t) {
            t.printStackTrace();
            throw SmartFrogException.forward(t);
        }
  }

  public void sfTerminateWith(TerminationRecord tr) {
        try {
            // close down everything
            sfLog().out("SmartFrog parent process terminating");
        } catch (Exception e) {}
```

```
        super.sfTerminateWith(tr);
  }
}
```

Example 17. Parent.java

When this configuration is deployed, only the **Parent** component is started. However, **Parent** can access the description of **Child** from within its local context. The Java class for **Parent** is given in Example 17.

The key steps are that **Parent** finds the description of the **Child** component by searching for the value of “myChild”, and then deploys it. The first step is in the **sfDeploy** method:

```
child = (ComponentDescriptionImpl)sfResolve("myChild");
```

Example 18

This call finds the parsed description of the **Child** and returns it as a **ComponentDescription** object as defined in the previous section. The second step is in the **sfStart** method:

```
sfCreateNewChild("Child", child, null);
```

Example 19

This deploys the **ComponentDescription** as a new child of the calling component. The first argument is the name of the new component. The second is the **ComponentDescription**. The last is a **Context** containing additional attribute/value pairs to add to those already defined for the child component. This makes it possible to carry out late binding to values not available at parse time, including values created within the parent object. In this particular case the **Context** is null. The code for the **Child** component is almost identical to the **Parent** object – it is a minimal SmartFrog component and can be found in Example 20.

```
// Child.java

package org.smartfrog.examples.tutorial.recipes;

import java.rmi.*;
import org.smartfrog.sfcore.common.*;
import org.smartfrog.sfcore.prim.*;

public class Child extends PrimImpl implements Prim {
  public Child() throws RemoteException {
  }
```

```
  public void sfDeploy() throws RemoteException, SmartFrogException {
        try {
            super.sfDeploy();
            sfLog().out("SmartFrog "+sfCompleteName()+" deployed");
            sfLog().out("SmartFrog child process deployed");
        } catch (Throwable t) {
            t.printStackTrace();
            throw SmartFrogException.forward(t);
        }
  }

  public void sfStart() throws RemoteException, SmartFrogException {
        try {
            super.sfStart();
            sfLog().out("SmartFrog child process started");
        } catch (Throwable t) {
            t.printStackTrace();
            throw SmartFrogException.forward(t);
        }
  }

  public void sfTerminateWith(TerminationRecord tr) {
        try {
            // close down everything
            sfLog().out("SmartFrog child process terminating");
        } catch (Exception e) {}
        super.sfTerminateWith(tr);
  }
}
```

Example 20. Child.java

11 References

The simplest content of a SmartFrog description consists of an attribute list – that is, a collection of attribute/value pairs. Each attribute consists of an attribute *name* and a attribute *value*. The syntax permits a **Reference** to stand in place of either a name or a value. There are three kinds of reference in SmartFrog:

- Placement references: the reference defines a context and an attribute *name*.
- Link references: the reference defines a context and an attribute *value*.

- Prototype references: the reference imports a prototype definition into the current context as the value of an attribute.

A reference consists of a colon-separated list of reference parts. A good way to think of reference parts is to visualise the tree of component descriptions described in Section 9.1; a reference part is a directive for navigating the tree of descriptions. The **Reference PARENT:count** refers to the attribute **count** in the context one level higher in the tree. This syntax can be likened to the familiar **../count** pathname syntax in UNIX and the analagous **..\count** used in MSDOS.

The complete syntax and semantics for references can be found in the *SmartFrog Reference Manual*, and there is no need to cover the same ground here, as they are adequately described. It is better to concentrate on some of the more common applications for this syntax.

One of the most common is the ATTRIB WORD syntax. It is often the case that a prototype description is intended to link into a larger context. The description in Example 21

```
#include "org/smartfrog/components.sf"
ResourceManager extends Prim {
   sfClass "org.smartfrog.examples.tutorial.recipes.ResourceManager";
   poolSize TBD;
}
```

Example 21

contains an attribute **poolSize** which is not defined. The **ResourceManager** needs to know how many resources to make available, but this value either cannot be defined at the level of the prototype description, or it must be defaulted to a value. The TBD syntax ensures that it *has* to be defined – this is better than using a potentially erroneous default value. The following deployment description contains two prototype references, both of which refer to the **ResourceManager** prototype defined above.

```
#include "org/smartfrog/components.sf"
#include “a file containing prototype descriptions”

sfConfig extends Prim {
   size 10;
   rm1 extends ResourceManager {
      poolSize ATTRIB size;
   }
   rm2 extends ResourceManager {
      poolSize ATTRIB size;
   }
}
```

Example 22

The attribute **poolSize** has been defined as **ATTRIB size**. This is resolved by moving up the tree of nested contexts until the attribute **size** is found. Both instances of **ResourceManager** are using the same value for **poolSize**; that is, 10. In this example **ATTRIB size** is semantically equivalent to **PARENT:size**. It could, in other cirmustances, be equivalent to **PARENT:PARENT:size** or **PARENT:PARENT:PARENT:size.**

The SmartFrog SDK contains classes **Reference** and **ReferencePart**. These can be used with the **sfResolve** method defined in **PrimImpl** to access values in other contexts in real time. This can be a good thing and a bad thing.

It is most definitely a bad thing when it is used to extract a value from a remote context that could just as easily be defined in the local context using references as described above. If an environmental dependency can be resolved at the time a component description is parsed, then it should be. A prototype describes a component in isolation from its environment, and a SmartFrog description provides references as a way to expand and link a prototype to its environment.

Comment:

The syntax

size PARENT:ATTRIB size;

will also work in Example 23. The potential circularity of an attribute **size** referring to an attribute **size** is avoided with an explicit initial reference to the parent context.

Some references may not be resolvable at parse time. These are tagged as LAZY. This is the most important use for constructing and resolving references at runtime. An example of this is given in the following section.

12 Discovery & Communication

It is often the case that one component will need to communicate with another component using Java RMI. This is easy to achieve. The SmartFrog configuration in Example 23 shows how it is done.

```
#include "org/smartfrog/components.sf"

Component1 extends Prim {
   sfClass "org.smartfrog.examples.tutorial.recipes.MyComponent";
}
Component2 extends Prim {
   sfClass "org.smartfrog.examples.tutorial.recipes.YourComponent";
   component1 TBD;
}
sfConfig extends Compound {
   myComponent1 extends Component1;
   myComponent2 extends Component2 {
      component1 LAZY ATTRIB myComponent1;
   }
}
```

Example 23

In this example, **component2** needs to communicate with **component1**. In the prototype definition **Component2**, there is an attribute **component1** which is declared but not defined.

The deployment consists of two components, **myComponent1** and **myComponent2**, based on the prototypes for **Component1** and **Component2**. The value of the attribute **component1** is now defined by a LAZY reference to **myComponent1**. What this means is that when the value of the attribute **component1** is requested at runtime, a Java remote interface for **myComponent1** is returned, and Java RMI calls can be made.

Let us suppose that **Component1** implements **Remote** Java interfaces **Errors** and **Management**. Then the code to access these interfaces from within **Component2** is shown below in Example 24.

```
// find the remote interfaces
Errors errorInterface = (Errors)sfResolve(“component1”);
Management managementInterface = (Management)sfResolve(“component1”);

// call methods on the remote interfaces
errorInterface.aRemoteMethodCall();
managementInterface.anotherRemoteMethodCall();
```

Example 24

The value of “component1” can be cast to any of the **Remote** interfaces implemented by **Component1** – in this case, **Management** or **Errors**. It is a common mistake to cast the value to the class **Component1**, rather than the **Remote** interfaces it implements. This will cause a **ClassCastException**. The interface returned by **sfResolve** *must* be cast to a *remote interface* implemented by the remote object.

Sometimes the location of a component is not known when a SmartFrog configuration is parsed and deployed, and the above method for obtaining a Java RMI interface cannot be used. There is an alternative that works when the IP address of the computer the component is running on is known, along with the SmartFrog name of the component (**sfProcessComponentName**). A Reference can be created using the HOST reference part construct.

```
String ipAddress = “15.1.2.3”;
String componentName = “FOOBAR”;

Reference aComponentRef = new Reference("HOST \""+ipAddress+"\":"+
componentName, true);

Foobar foobar = (Foobar) sfResolve(aComponentRef);
```

Example 25

If it is necessary for a SmartFrog component to provide information about itself to another component, then Example 26 provides useful code fragments.

```
// find the fully qualified host name of this host
String hostname;
hostname = ((InetAddress)sfResolve("sfHost")).getCanonicalHostName();

// find the IP address of this host
String ipaddress;
ipaddress = ((InetAddress) sfResolve("sfHost")).getHostAddress();

// extract a Remote interface for this RMI server
Remote interface;
interface =  RemoteObject.toStub(this);
```

Example 26

13 Conclusion

This tutorial has only touched on the ways that SmartFrog can be used. The SmartFrog distribution contains many complete examples under src/org/smartfrog/examples. These examples include the prototype descriptions, the deployment descriptions, and the corresponding Java components. One of the intentions behind this tutorial was to make it obvious what these examples are intended to achieve. Studying these examples is strongly recommended. It is also worth studying the Java for some of the major classes, such as **PrimImpl** or **CompoundImpl**. It should become clear that the default behaviours were designed to be extended.

More detailed information on many aspects of SmartFrog programming can be obtained by referring to:

The SmartFrog Reference Manual

The SmartFrog User Guide

The SmartFrog API (Javadoc)

The SmartFrog Quick Reference Guide

14 Acknowledgments

Thanks to the SmartFrog team for much assistance and to Patrick Goldsack and Julio Guijarro for their comments on the draft of this document.

15 Tutorial References

[1] See “sfInstaller” component and templates for examples on how to install, start and stop SmartFrog daemons in a cluster of machines.

16 Appendix 1 – Defined Attributes

The SmartFrog language contains a number of reserved words, some of which are used to set values, others are used to discover values. An extended list can be found in the *SmartFrog Reference Manual*.

sfProcessHost: this specifies a host where a component will be deployed. Eg.

```
myComponent extends aComponent {
   sfProcessHost “server.hp.com”; // where the component will run
}
```

Example 27

sfProcess: name of the process in which a component is running. Always set.

sfProcessName: name of the process you want to deploy in. This could cause a sub-process to be created.

sfProcessComponentName: set the name of a component in the root process or a sub-process.

```
myComponent extends aComponent {
   sfProcessComponentName “MYCOMPONENT”;
}
```

Example 28

sfRootLocatorPort: set the RMI port (default 3800) of the SmartFrog root daemon. Used only in a ProcessCompound.

sfHost: the **InetAddr** of host. An example of its use is given below:

```
// find the fully qualified host name of this host
String hostname;
hostname = ((InetAddress)sfResolve("sfHost")).getCanonicalHostName();

// find the IP address of this host
String ipaddress;
ipaddress = ((InetAddress) sfResolve("sfHost")).getHostAddress();
```

Example 29

17 Appendix 2 – Tracing and Logging

SmartFrog is capable of producing large quantities of useful diagnostic output. The first configuration file one should be aware of is `default.ini`, which can be found in `SFHOME/bin`. This contains an attribute that regulates the severity of logging information written to the output window (or logfile) associated with the SmartFrog daemon. The relevant portion of `default.ini` can be seen in Example 30. Note that you can also configure logging to a file, and other logging attributes.

```
# Optional property to define the initial local log level
#   Default level: 3 - LOG_LEVEL_INFO;
# 0-IGNORE(ALL);1-TRACE;2-DEBUG;3-INFO;4-WARN;5-ERROR;6-FATAL;7-NONE
    org.smartfrog.sfcore.logging.LogImpl.logLevel=0

# Optional boolean property to include stack trace with error message
   org.smartfrog.logger.logStackTrace=true
```

Example 30

A second file, also in the `SFHOME/bin` directory, is `default.sf`. This defines a number of prototypes, and deploys the default output component for the SmartFrog daemon process. There is an alternative commented-out deployment that provides enhanced lifecycle event monitoring, and the ability to step through the stages of a deployment. The relevant portion of `default.sf` is:

```
sfConfig extends DefaultCompound;


// Alternative default compound with tracing or/and log to file
enabled.
// Uncomment the next line to use it.
//sfConfig extends DefaultTraceCompound;
```

Example 31

If security is disabled, `default.ini` and `default.sf` may be edited in place. If security is enabled, then they are accessed from JAR files, and the JAR files will need to be recreated whenever modifications are made.

A tool that is often useful is the SmartFrog Management Console. This is described in the *SmartFrog User Manual*. The console provides a view of a deployed application, and provides the attribute and value in the context of each deployed component. For numerous reasons there may be some confusion about what has been deployed – it is easy to duplicate configurations in JAR files, it is easy to override values by extending prototypes. The console provides a quick way to check what is deployed, what attributes have been defined, and what their current values are.