1.SmartFrog WorkFlow

DRAFT
- WARNING: THIS IS SUBJECT TO CHANGE -

http://www.smartfrog.org

HP Laboratories, Bristol
Last revision: 2004-06-09

Introduction

This document shows how the SmartFrog system has been extended to become a simple and lightweight workflow-like system (known as SmartFlow) for carrying out complex tasks requiring distributed actions on clusters of machines, and where ordering, recovery from failure, etc. are important features.

The system is very much in its infancy and largely experimental, so the constructs provided here may be inadequate in some significant ways. However, as new components or variants of semantics are identified, these can easily be programmed and added to the SmartFlow system. At a few points in the document, issues and problems with the workflow framework are highlighted. If others are identified or additional functionality desired, please inform the SmartFrog team.

The SmartFlow system provide some of the core features for distributed scripting of synchronized actions, as required by a workflow system. Some key components for a full-blown system are missing – such as transactions and persistence - but concepts such as parallel and sequential composition of distributed tasks are provided.

The document covers a number of aspects:

1. How SmartFrog is a suitable framework for such as system.
2. The features that should be provided.
3. The design of the SmartFlow system itself and details of its components.

4. An example showing some of the features in action. This example is supplied with the distribution of the system.

The document assumes that the reader has a good understanding of the SmartFrog system.

Core SmartFrog Concepts

SmartFrog is a notation and supporting infrastructure for the development of components for instantiation on potentially remote processors. The current SmartFrog supports two kinds of component – the *Primitive* component (which is the user-defined components that carry out some activity for the system), and the *Compound* component which is a grouping of other components, be they compound or primitive. The sub-components may be started on whichever set of processors of a cluster they needs to be started – this association between component and processor being an attribute of the component itself. Each sub-component may be on a different processor if necessary.

The core semantics of compound are that

- on creation, its sub-components are also created
- if any of the sub-components terminate, so does it
- and similarly for a number of lifecycle stages

This semantics makes sense if we are trying to define a notion of an atomic collection – all of which is in the same state (created, idle, running, terminated, etc.). This must be extended if we are to have the concept of sequential creation of components, for example.

Fortunately, SmartFrog is a framework – it has been designed to allow further types of component to be defined and incorporated into the system. In particular it is possible to introduce new types of collections of sub-components with varying semantics suiting different target systems. Thus this document described a number of new component types that allow the definition of workflow-style applications. This is done without in any way modifying the SmartFrog notation or system, merely introducing some new classes of component in a way the framework already enables.

Workflow

Consider the following system management task. There is a cluster of machines, each running a version of a daemon. The system manager needs to update the version of this daemon on all machines, or if he fails he must rollback to the previous version. The system manager must carry out the following sequential steps:

- save a copy of the old software, and old daemon configuration on each machine;
- download the new software to each machine;
- restart all the daemons;
- if any failure occurs – replace the old software and if necessary restart the old daemon.

The identical actions on the separate machines may all be done in parallel, whilst the individual steps must be done sequentially – completing each before moving to the next. It is this combination of sequenced and parallel activities that is termed, for the purposes of this document, a *workflow*.

It is clear from the example that there are several features that make SmartFrog a suitable framework for progress

- there is the notion of component, specific ones may be provided to carry out a specific task such as moving files, restarting a daemon, etc.;
- there is the notion of composition (though with different semantics to Compound);
- there is the notion of remote creation of components on remote machines.

Thus if we can define a new set of composite components, such as sequence, parallel, etc. we can create workflows of primitive or compound components to achieve complex configuration procedures.

Event Framework

SmartFlow requires the provision of an event framework. The framework is a very simple distribution mechanism for events that hides from the components from where events are to be received or who are to receive any events they generate. The framework is slightly misleadingly called an event bus as though it differs significantly from the more traditional meanings of the term.

The SmartFlow event framework consists of a graph of component nodes and event forwarding arcs. Events are passed between components along the arcs of this graph, potentially along multiple paths, until they reach their end-points. At any intermediate point, a component may chose to act on the event or to ignore it acting merely as an event forwarder.

There are a number of new interfaces associated with the event framework.

1. **EventBus** – the public interface used by components that defines the template and API methods for components using events.
2. **EventRegistration** – an internal interface provided for use by the SmartFlow system for carrying out registrations for event forwarding.
3. **EventSink** – an internal interface provided for use by the SmartFlow system for forwarding events from one component to another.

The *EventBus* interface is the only one that needs further explanation. The interface is implemented by two new Java classes, *EventPrimImpl* and *EventCompoundImpl*, and these may be used in place of *PrimImpl* and *CompoundImpl* if a component needs to be part of the event framework.

The *EventBus* interface provides two methods:

```
void handleEvent(String event)
void sendEvent(String event)
```

The first of these is a template method; i.e. one that is pre-defined with a default action (ignore the event) but is available for the component to overwrite if some action is desired. It is called whenever an event arrives at the component. Note that this method is called in addition to forwarding the event to all registered components.

The second, *sendEvent*, is an API method and should be called by the component whenever it wishes to send an event into the system.

Events are strings. They may have structure beyond that, in that the strings may be used by a receiving component, and pattern-matched to extract internal structure. However, the event framework is unaware of this structure and does no filtering.

Forwarding relationships are defined in the SmartFrog descriptions, using links, and the *EventPrims* and *EventCompounds* automatically set up the forwarding graph. The key to defining this graph is the definitions of the *EventPrim* and *EventCompound* component descriptions. These are:

```
EventPrim extends Prim {
    sfClass "org.smartfrog.sfcore.workflow.eventbus.EventPrimImpl";
    sendTo  extends LAZY {}
    registerWith extends LAZY {}
}

EventCompound extends Compound {
    sfClass "org.smartfrog.sfcore.workflow.eventbus.EventCompoundImpl";
    sendTo  extends LAZY {}
    registerWith extends LAZY {}
}
```

As can be seen, they both have *sendTo* and *registerWith* LAZY components. These may contain LAZY links to other components that implement the event framework and define the graph of event forwarding. Note that the registration can be done in both directions, i.e. deciding which components a component wishes to receive from or the other way around. This provides a great deal of flexibility in providing the configuration for the event-forwarding graph.

Every SmartFlow component implements the event framework and may therefore be used to forward events around the system. An example of the use of events is given in the worked example at the end of the document.

Note: The event framework is a first pass at the concept, and is not without its problems. In particular, it is sometimes hard to find out how to synchronize parts of the workflow because inevitably there is a race condition between the event being generated (say the creation of a *SendEvent*, see below) in one sequence and the creation of the interested component (for example the creation of an *OnEvent*, see below) in another sequence. Events are transient, so if the event is generated before the registration of interest, it is lost forever. This will be fixed in the next release of the framework – a number of solutions are being evaluated.

Currently the solution lies in providing top-level receivers for events, created before the generation, that act as schedulers for the rest of the workflow; these must currently be provided by the user. This concept is one of the generalised solutions to the problem being investigated, as is caching events within the system for forwarding to interested components as registrations occur.

SmartFlow

The primary purpose of this section is to define the various components that may be used for creating workflows. This will be done in two parts – first showing the overall list of components provided to get a feel for the scope of SmartFlow, the second describing in detail how these are used.

The List of Components

1. **Parallel** – to provide the simultaneous execution of multiple sub-components. The parallel component only terminates when all sub-components independently terminate.
2. **Sequence** – starts the sub-components one at a time, starting the next when the previous one terminates.
3. **Timeout** – waits a certain period for the completion of its sub-component, if not terminated in that time – fails itself and its sub-component. Early termination propagates the sub-components failure mode.
4. **During** – waits for a certain period of time after which it terminates its sub-component if not already terminated – terminates itself normally at that point. Early termination propagates the sub-components failure mode.
5. **Retry** – tries to execute its sub-component, if it fails, tries again and so on for some maximum number of retries with some time-delay between retries.
6. **Repeat** – executes its sub-component. If the sub-component terminates normally, repeats this a given number of times.
7. **Try** – tries to carry out its primary sub-component. When it terminates it will invoke the appropriate sub-component indexed by termination code. Its normal use is for rollback after failure, with the ability to invoke some further action even after normal termination such as issuing a notification of success.
8. **Delay** – wait for a period and then proceed with its sub-component. Termination of the delay component is the same as that for its sub-component.

In addition to these scripting components, there are others required for starting and stopping other applications as part of the workflow process. For this there are two components provided:

1. **Run** – starts its sub-components as independent agents, then terminates normally, not remaining the 'parent' composite component for the subcomponents. It is used for launching long-term services, using the workflow part to ensure the appropriate conditions have been achieved, and that the agents are started in the right order.
2. **Terminator** – given a reference to an application, call sfTerminate upon it with an appropriate termination type, then terminate self.
3. **Attribute** – given a reference to an component, and the name of an attribute within it, add, replace or remove that attribute as appropriate.

These composites allow the construction of complex sequencing and interleaving of actions, however this is not enough on its own. There will in general be the need to synchronise components from different parts of the workflow. This can be done using the event framework (modulo the issues described above).

To simplify the use of the event framework and synchronize the various parts of the workflow, a number of components are provided:

1. **OnEvent** – a compound component that registers with the event framework (if necessary), waits for an event and dispatches an appropriate sub-components according to event structure.

2. **EventSend** – a primitive component to send a specific event to the event framework, perhaps to a waiting EventCounter or OnEvent, and terminates. Typically it is used in a sequence to signal progress to a parallel sequence.
3. **EventCounter** - a dual of EventSend is the EventCounter, a component that counts the events it receives and when it reaches a trigger level, terminates. It can be used in a sequence to pause until some other tasks have been completed.
4. **EventQueue –** a buffer of events, forwarding all the events it has received to any component that registers – even if this is after the event has been received. This allows for the synchronization of seperate sequences of actions.

Finally, a do-nothing component is required, if only as commentary or for test purposes.

1. **DoNothing** does more-or-less absolutely nothing for a period and terminates in a way defined by its description. Also provides event reciept and message printing, so is very useful for debugging and tracing of workflows,

Detailed Description of Components

The components will now each be described in detail, in roughly reverse order of grouping. They all have some similarity in that

- they all leave the lifecycle of the defined templates until their own *sfStart* phase, at this point they choose the necessary templates (defined as LAZY component descriptions) and create, initialise and start the cub-component.
- all (but *Parallel* and *Sequence*) use the template defined in the attribute *action* to determine the initial sub-component that must be created
- *Parallel* and *Sequence* use the templates defined in the component description *actions.* This is because they need more than one template. Parallel steps all the templates synchronously through their lifecycle, Sequence only steps the first.
- all (but Parallel) create their child sub-component under the attribute name *running*. All subsequent components are also called *running* since they are only created after the termination of the previous ones.
- *Parallel* names its children after the names they had in the *actions* component description, this is because more than one runs at a time, and so a single name is insufficient.

Note: many components are compounds, so will deploy any normal sub-component as is usual for *Compound*. However, the components described here will get very confused if any of these terminate, so this facility should only be used if the sub-components are guaranteed not to terminate or fail their lifecycle. Consequently, they should not be remote. This will be fixed in the next release of the framework.

DoNothing

```
DoNothing extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.DoNothing";
```

```
    time 10000000;                    // time in milliseconds to wait
    terminationType "normal";         // termination record type
    printEvents false;
    // message "a message to print on the console prefixed by full name"
}
```

The do-nothing construct does nothing for a time, then terminates as required. Its purpose is primarily to act as a stub for bits of workflow yet to be written during testing. If *terminationType* is set to *"none"*, it waits for ever.

The component prints the message to stdout and starts the timer during its start phase and terminates when the timer has fired. If the time is 0, the component will terminate immediately – though a thread is started to do this so some asynchrony occurs. If no message is provided, no message is printed.

If the attribute *printEvents* is set to *true*, any events that it receives will be printed on the console. If *printEvents* is *false*, it does nothing internal with events, merely forwarding them as required.

EventSend

```
EventSend extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.EventSend";
    // event "the string to send as an event";
}
```

An event sender contains an event to send, and it forwards the event to any registered components as defined in the event framework.

The component sends its event during the start phase, and then terminates asynchronously. It does nothing internal with received events.

EventCounter

```
EventCounter extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.EventCounter";
    count 1;
    allDifferent "true";
}
```

Event counter waist for events, when they arrive it decrements its counter (default 1). When the counter reaches 0, the component terminates. If the attribute allDifferent is set to true, the counter is not decremented if the event it receives is an identical string.

OnEvent

```
OnEvent extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.components.OnEvent";
    //event1 extends LAZY ... { ... }
    //event2 extends LAZY ... { ... }
    //otherwise extends LAZY ... { ... }
}
```

The OnEvent event dispatcher provides a number of named event handlers as attributes – defined as LAZY components. When an event arrives, it is compared to the names of the attributes using Java string equality, and if a matching attribute is

found the component description is taken and deployed. If no match is found, the *otherwise* template is taken. If that is not present, the component terminates with an error.

Once a handler has been started, it runs as the newly created sub-component as a child using the name *running*. When that child terminates, so does the OnEvent, with the same termination type.

EventQueue

```
EventQueue extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.EventQueue";
}
```

The EventQueue component forwards events to registered components as all other workflow components do. However, there are two additional differences:

1. The events are stored and any future registrations will receive all the events in the order in which they arrived at the EventQueue, not only new ones. This makes the EventQueue a suitable component to provide the synchronization point between two sequences where one cannot be certain that the event receiver will be ready (or exist) before the sender is required to send the synchronization event.
2. The event forwarding is asynchronous. In other words, the thread that delivers the event to the Queue returns before the events are necessarily forwarded. This is not so of the default event forwarding which is synchronous. Thus an EventQueue can be used wherever it is required to provide that level of asynchrony.

Run

```
Run extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Run";
    // action extends LAZY ...;
    // parent ref to optional parent compound
    // asName  a string to be the name in the parent compound
}
```

Run is a component that, during its start phase locates the action attribute – which must be a component description provided as a LAZY component – and deploys it as a separately running application. Once this is done, the run component terminates asynchronously. It fails if the application has not been started correctly.

The launched application would normally name itself on some host so that it may later be found – normally using the sfProcessComponentName attribute.

Alternatively, it is possible to provide a link to a parent compound and the name the component should have in that compound. This is done using the attributes parent and asName as defined in the prototype above.

Terminator

```
Terminator extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.Terminator";
    // kill LAZY RefToComdemnedComponent;
    type "normal";
    description "terminator action";
    detachFirst false;
}
```

Terminator is a component that, during its start phase, de-references its kill attribute to find a component (it is an error if it does not do so). It then constructs a termination record from the type and description attributes and invokes the sfTerminate method on the referenced component.

Frequently, the reference will be a host reference, followed by the application name as set by the application's top-level sfProcessComponentName attribute.

If the component is a sub-component of a compound that should not be terminated, the component may first be detached. This is done by setting the detachFirst attribute to true.

Attribute

```
Attribute extends EventPrim {
    sfClass "org.smartfrog.sfcore.workflow.components.Attribute";
    /*
    component RefToComponent;
    name "a string";
    value newValue; // not present implies remove...
    */
}
```

Attribute replaces, removes or adds an attribute of a primitive or compound component as defined by the attributes provided. The component is a reference to the component containing the compound. The name indicates the name of the attribute in that component. The value, if present, determines the value this attribute should take, or if absent defines that the attribute should be removed.

Parallel

```
Parallel extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Parallel";
    actions extends LAZY {}
}
```

Parallel is the closest SmartFlow comes to the standard Compound. It differs in a number of ways.

Firstly, to conform to the structure and lifecycle of all the other SmartFlow components, the sub-components are all defined in the *actions* component description that must be LAZY so that no component is automatically created. The component templates may be placed here, or the *actions* attribute may be redefined when extending Parallel.

Secondly, the sub-components are only created, initialised and started during the *sfStart* phase of the parallel, rather than during its equivalent phase. Again, this is to conform to the general structure of SmartFlow.

Thirdly, the parallel waits until each of the sub-components terminates before terminating itself. This is true, at least, for normal termination. Any abnormal termination is considered fatal and the whole parallel and its remaining sub-components will be terminated. This is also true of any forced termination by a non-child component.

Note that the parallel components have their lifecycle phased with respect with each other, so during their initialisation they may reference each other, locate each other, and so on. Note also, that once created they sub-components are children of the Parallel component itself, and are named using the names of their in the *actions* component description. LAZY references from elsewhere should therefore take cognisance of this rather than assuming the sub-components will be within the *actions* component description.

Note also that *Parallel* may be used directly with the normal sub-components (i.e., not lazy and not defined as templates in *actions*) just as in Compound, phasing lifecycle steps with those of the *Parallel*. Even if used in this way, *Parallel* still implements independent termination. The provision of the *actions* attribute, and the limitations of the lifecycle to *Parallel*'s *sfStart*, is primarily for conformance with the rest of the framework

Sequence

```
Sequence extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Sequence";
    actions extends LAZY {}
}
```

A sequence takes each component through its entire lifecycle, one at a time. As the first terminates, the second is created and starts. The templates for these components are found in the actions component description, and they are created, initialised and started in the order given in the component description.

The first component is created, initialised and started during the *sfStart* of the sequence. It is then left to run to completion, at which point the next template is taken and a component created and started. This continues until one of the components terminates abnormally or all the components have completed their lifecycles.

When they are running, the sub-components are all called *running*, and as only one at a time exists, this is possible.

Note that it is frequently useful to have components that exist at the level of the Sequence that persist throughout its lifetime, perhaps to act as a persistent data store for the sequnce. This is possible as the Sequence is also a normal compound, and components may be added as child components in the normal way. These will be terminated by the Sequence when the last of the actions is complete.

During

```
During extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.During";
    // action extends LAZY …
    time 0;                          // time in milliseconds
}
```

During runs a sub-component for a period of time and, if it has not terminated in that period, terminates it and itself normally. The sub-component to launch is defined by the *action* component description that must be a LAZY description. It is created and started in the *sfStart* phase of *During*. The default time is 0, indicating that it should terminate immediately. The time is given in milliseconds.

Note that the sub-component, when it is created, is known as *running*.

Timeout

```
Timeout extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Timeout";
    // action extends LAZY …
    time 0;
}
```

Timeout runs a sub-component for a period of time and, if it has not terminated in that period, terminates it and itself abnormally. The sub-component to launch is defined by the *action* component description that must be a LAZY description. It is created and started in the *sfStart* phase of *Timeout*. The default time is 0, indicating that it should terminate immediately. The time is given in milliseconds.

Note that the sub-component, when it is created, is known as *running*.

Delay

```
Delay extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Delay";
    // action extends LAZY …
    time 0;
}
```

Delay starts a thread during its *sfStart* phase that delays a period of time, then creates a sub-component as a child of *Delay*. This child has the name *running*. Delay terminates when the child terminates with the same termination type. The default time is 0 and is given in milliseconds.

Retry

```
Retry extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Retry";
    // action extends LAZY …
    retry 1;
}
```

Retry creates and starts a sub-component during its sfStart phase, as defined by the *action* template, and waits for its termination. Whilst it is running, the child sub-component has the name *running*. If it terminates normally, so does *Retry*. If it terminates abnormally, *Retry* will start another instance of the action template, again called *running*. This continues until either the child terminates normally, or the specified number of reties is exhausted. The default number of retries is one.

Repeat

```
Repeat extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Repeat";
    // action extends LAZY …
    repeat 1;
}
```

Repeat creates and starts a sub-component during its sfStart phase, as defined by the *action* template, and waits for its termination. Whilst it is running, the child sub-component has the name *running*. If it terminates abnormally, so does *Retry*. If it terminates normally, *Retry* will start another instance of the action template, again called *running*. This continues until either the child terminates abnormally, or the specified number of repeats is exhausted. The default number of repeats is one.

Try

```
Try extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Try";
   // action extends LAZY …
   // normal extends LAZY …
   // abnormal extends LAZY …
   // etc
}
```

Try creates and starts a sub-component during its *sfStart* phase, as defined by the *action* template, and waits for termination. The running child has the name *running*. When termination occurs, the termination type is examined and *Try* finds an attribute whose name is the same as the termination type. If it does so, it creates and starts a sub-component as defined by the template attribute. If it does not have a continuation, *Try* terminates by propagating the termination of the initial child component. When the continuation component terminates, *Try* does so as well by propagating the termination.

References

As the component tree of a SmartFlow hierarchy is built during the sfStart phase, or afterwards triggered by the termination of a component or the arrival of an event, most components will not be available all the time. Indeed, some components will never be created.

Consequently, assumptions that are usually made about the existence of components during initialisation are no longer valid. In some cases, the components may well be – for example in each branch of a parallel. However, the general rule certainly does not apply.

The trick is in general only to use references that go up the tree – these are always valid. Those referencing down the tree may not be valid, detailed knowledge of the lifecycle is important when using these.

Note also, that the majority of sub-components created by the various components are called “running” within the parent. The main exception to this is the parallel, where the children are known by the appropriate names within the *actions* component description.

Termination

Since SmartFlow is designed to carry out a series of tasks, each more or less short-lived, it is tempting to place the entire logic of the component to carry out a task in the *sfStart* method. This is in general OK, however termination of the component may not be directly initiated from inside the *sfStart* method, i.e. *sfTerminate* may not be called in *sfStart* as it blocks under certain circumstances.

Consequently, if termination is required at the end of the sfStart method, add the following snippet of code to trigger asynchronous termination.

```
Runnable terminator = new Runnable() {
     public void run() {
          sfTerminate(term);
     }
};
new Thread(terminator).start();
```

This starts a new thread to trigger termination, allowing the sfStart method to complete normally.

It is likely the future versions of SmartFrog will provide components that are better suited to SmartFlow, but for now the underlying system assumes that termination is always triggered asynchronously.

Example

There is a simple example of the use of the SmartFlow system provided with the distribution – contained in package *org.smartfrog.examples.workflow*. This example is listed here with little addition commentary.

The example is of running a number of applications for a short time period on a set of hosts. Each application consists of two parts, a web server and an application server. Before these applications can be run, some preparatory actions must be taken – downloading some files – and on termination, this must be undone to clean up.

Note that the application is not real in that all the actions are stubs, simply printing their intended behaviour to the console of the daemon in which they are running.

The example is divided into four files, building from the bottom up.

- base.sf contains the basic building blocks for the application
- application.sf uses these to define the notion of an application as having a preparatory phase, and action, and a clean-up phase. Two application are defined – the web server and the application server.
- node.sf defines what should run on a particular node – namely a web server and application server. It also defines the notion of a reliable node, one that retries the applications for a number of times until successful
- system.sf contains the final system, with five hosts, and their mappings to the different physical servers for execution.

To make the example more interesting, in addition to a well-behaved application, one instance of the application has an additional component added which deliberately kills it, thus causing the reliability features to kick-in.

File base.sf:

```
#include "org/smartfrog/components.sf";
#include "org/smartfrog/functions.sf";
#include "org/smartfrog/services/workflow/components.sf";
#include "org/smartfrog/services/scripting/components.sf";


/*
 * Simulate a number of basic actions - such as
 *      copying and removing files
 *      running daemons
 *
 * These are simulated (view them as testing stubs for the overall logic)
 * using the DoNothing component and the scripting capability.
 */



/* The copyfile component is given a URL of a file,
 * and the name of the file to create,
 * issues a message stating that the file has been
 * copied, delays a couple of seconds, and terminates.
 */

Copyfile extends DoNothing {
    time 2000;

    //default URL to copy to file
    fromURL "http://defaultHost/defaultFile";

    //default name for file
    toFile "/tmp/default";

    message extends concat {
        a "copied file from ";
        b ATTRIB fromURL;
        c " to file ";
        d ATTRIB toFile;
    }
}


/*
 * The removefile component is given a filename to remove,
 * simulates doing so by issueing a message to that fact,
 * then terminates after two seconds.
 */

Removefile extends DoNothing {
    time 2000;                              // time in millisecs
    file "/tmp/default";                    // default name for file

    message extends concat {
       a "file ";
       b ATTRIB file;
       c " has been removed";
    }
}


/* There are two choices for the daemon
 * either we can have a component that starts it then terminates
 * and then another that stops it or we can launch a component
 * that exists whilst the daemon should be running, starting in sfStart
 * and terminating in sfTerminateWith.
 * We will do the latter using ScriptPrim to provide the simulation
 * by issuing messages at the right point.
 *
 * A complete implementation for this would use the runCmd component
 * in .../os/runCmd
 *
 * The component is given the command line as a string
 */

RunDaemon extends ScriptPrim {
    commandLine "echo running daemon";
```

```
    sfStartCode ##
       System.out.println("starting " + prim.sfFind("commandLine"));
    #;

    sfTerminateWithCode ##
       System.out.println("stoping " + prim.sfFind("commandLine"));
    #;
}
```

File application.sf

```
#include "org/smartfrog/examples/workflow/base.sf"

/*
 * define an application to be a sequence of
 * copy file; run command for a period of time; remove file;
 *
 * running a command for a period involves wrapping in "during"
 * to limit the time of the run
 *
 * an application is parameterised by the file URL, filename
 * and the commandline, plus the length of time to run the daemon
 *
 */


Application extends Sequence {
    commandLine "echo default command";
    filename "/tmp/default";
    fileURL "http://defaultHost/defaultfile";
    time 10000;                     // default time in millisecs
    sfSyncTerminate "true";

    actions extends LAZY {
       a extends Copyfile {
           toFile ATTRIB filename;
           fromURL ATTRIB fileURL;
       }
       b extends During {
           sfSyncTerminate "true"; // child stops before telling parent
           time PARENT:ATTRIB time;              //avoid a loop..!
           action extends LAZY RunDaemon {
               commandLine PARENT:ATTRIB commandLine; //avoid a loop..!
           }
       }
       c extends Removefile {
           file ATTRIB filename;
       }
    }
}



/*
 * Now to have some specific examples of applications
 *
 * a web server and an application server
 *
 */

WebServer extends Application {
    file "webServerCode";
    fileURL "http://codeserver/webServerCode";
    commandLine "httpd";
}

AppServer extends Application {
    file "appServerCode";
    fileURL "http://codeserver/appServerCode";
    commandLine "blustone";
}
```

File Node.sf

```
#include "org/smartfrog/examples/workflow/application.sf"
#include "org/smartfrog/services/trace/components.sf"

/*
 * define a node to be a sequence of
 *     a compound consisting of a web server and an app server
 *     notification of completion to a completion monitor
 */

Node extends Compound {
        sfSyncTerminate "true";
        webs extends WebServer;
        pps extends AppServer;
}



/*
 * to experiment with failures - we can simulate a
 * rogue node where everything fails ungracefully
 * by addding a component which terminates abnormally
 * after a very short time - say 4 seconds
 *
 */

RogueNode extends Node {
    failure extends DoNothing {
       terminationType "abnormal";
       time 4000;
    }
}



/*
 * we can now make a "reliableNode"
 * which is one that tries 3 times to launch
 * the correct action and if it fails sends a message
 * to the sys admin
 *
 * parameterised on the action and retries
 *
 */

NotifySysAdmin extends ScriptPrim {
    sfStartCode ##
        System.out.println("Houston - we have a problem with component ");
        System.out.println(prim.sfCompleteName());
    #;
}

ReliableNode extends Try {
    nodeAction extends LAZY DoNothing; // default action for reliable node
    retries 1;
    sfSyncTerminate "true";

    action extends LAZY Retry {
        sfSyncTerminate "true";
        action ATTRIB nodeAction;
        retry PARENT:ATTRIB retries;
    }

    // on abnormal termination, notify admin
    abnormal extends LAZY NotifySysAdmin; }


/*
 * Given this, we can now define a reliable system node and a rogue system
 * node as being the reliable node wrapping the node and reliablenode
 */

SysNode extends ReliableNode {
    nodeAction extends LAZY Node;
}

RogueSysNode extends ReliableNode {
    nodeAction extends LAZY RogueNode;
}
```

File system.sf

```
#include "org/smartfrog/examples/workflow/node.sf"
#include "org/smartfrog/examples/workflow/notifier.sf"

/*
 * Now define a system as containing a number of nodes,
 * say 4 normal and one rogue
 *
 * These are run in parallel to provide independant termination,
 * the system terminating when all the sub components do
 *
 * parameterize by their hostnames
 * set to localhost by default
 *
 */

System extends Parallel {
    host1 "localhost";
    host2 "localhost";
    host3 "localhost";
    host4 "localhost";
    rogueHost "localhost";

    actions extends LAZY {
       h1 extends SysNode {
           sfProcessHost ATTRIB host1;
       }
       h2 extends SysNode {
           sfProcessHost ATTRIB host1;
       }
       h3 extends SysNode {
           sfProcessHost ATTRIB host1;
       }
       h4 extends SysNode {
           sfProcessHost ATTRIB host1;
       }
       rh extends RogueSysNode {
           sfProcessHost ATTRIB rogueHost;
       }
    }
}


/*
 * deploy a system - setting host names as required
 *
 */

sfConfig extends System {
    host1 "localhost";
    host2 "localhost";
    host3 "localhost";
    host4 "localhost";
    rogueHost "localhost";
}
```

License

```
/** (C) Copyright 1998-2004 Hewlett-Packard Development Company, LP

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA

For more information: www.smartfrog.org

*/
```