SmartFrog Workflow

25 Oct 2007

1 Introduction

This document shows how the SmartFrog system contains a set of workflow components for carrying out sequential or complex tasks across one or more machines, tasks where ordering and recovery from failure are important features.

The workflow components provide some of the core features for distributed execution of synchronized actions, as required by a workflow system. In particular components can be deployed in parallel or sequentially, with special container components to manage complex lifecycles.

The document introduces the container components, and some of the components that can be used inside them. It also introduces the *conditional* components, which can be used to implement run-time logic. It assumes that the reader has a basic understanding of the SmartFrog system.

2 SmartFrog and Workflow

SmartFrog is a notation and supporting infrastructure for the development of components for instantiation on local and remote processors. 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 the `Compound` are that:

- On creation, its sub-components are also created
- If any of the sub-components terminate, it does too.
- Liveness checks, `sfPing()` operations, are passed down if the `Compound` is pinged by its parent -but not if the ping is sent by any other component.
- When the `Compound` is terminated, so are its children.

These semantics create the notion of a static deployment, in which components are deployed into a specific state, a state which holds for the life of the deployment. Individual components implement the startup and teardown operations for their specific library/application, and a simultaneous deploy/undeploy is exactly what is needed. This is an implicit "simultaneous deployment" workflow.

Large and complex deployments have more complex deployment processes, and need components that can describe them. Here are some common real-world use cases:

1. The MySQL JDBC JAR file needs to be downloaded from a remote repository, and copied to JBoss/server/lib, after which JBoss can be started.
   This can be represented by a sequence of actions: a download/copy followed by the execution of the JBoss process.
2. The MySQL database needs to be started, then configured with some tables and data. The startup time of MySQL is dependent upon the size of the database.
   This can be represented by a parallel deployment of the database and the SQL commands, commands which should be blocked until the database is actually visible.
3. At the end of a test run, a SQL command to drop a table should be issued.
   This can be represented by an action that takes place on system teardown.
4. Repeated attempts to copy a file should be made until the operation is successful; if the file is in use it will be locked and the copy will fail.
   Here we need the notion of an operation which can be retryed until it succeeds.
5. A SmartFrog application should be deployed on a remote machine, with an independent lifecycle. If the

remote machine is not responding, the attempt should be repeated -after a delay.

6. A system test consists of deploying a remote web site, waiting for it to be visible, then running a series of tests. Once the tests have finished, the web site application must be terminated.

Although custom components could be written to choreograph such sequences, this is inappropriate. Coding complex lifecycles into the components that perform the actual work leads to inflexible components. What we need is a *Separation of Concerns*, in which the rules of how multiple components are deployed are implemented in *Workflow Container* components, components that deploy their children in controlled lifecycles, lifecycles that can be driven by attributes of the container.

These containers can be aided by workflow components, components that can signal events to other components, and which can register to receive such events. This enables active components to await events from others, and so be triggered when the components they are listening to reach a certain state.

3 Event Framework

Workflow components have the ability to send events to each other, even remote nodes. The event distribution mechanism decouples the components from the sources and destinations, with a uniform interface for subscribing and publishing events. The framework is slightly misleadingly called an event bus as though it differs significantly from the more traditional meanings of the term.

The 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. **EventSink** – an internal interface for forwarding events from one component to another.
3. **EventRegistration** – an internal interface for carrying out registrations for event forwarding.

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(Object event)
void sendEvent(Object event)
```

The first of these is a template method; i.e. one that is pre-defined with a default action (ignore the event) which components can overwrite if some action is desired. It is called whenever an event arrives at the component. 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 serializable objects. They may have structure beyond that, in that the objects 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 `EventPrim` and `EventCompound` instances 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 {}
}
```

Both components have `LAZY` `sendTo` and `registerWith` components. These can contain late-binding links to other components that implement the event framework and define the graph of event forwarding. The registration can be done in both directions; senders can choose recipients while and receivers can choose who to listen to. This provides a great deal of flexibility in providing the configuration for the event-forwarding graph.

Every workflow container 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 distributed deployment example at the end of this 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.

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.

4 Workflow Components

With the theory completed, let us explore the components.

4.1 The Primary Workflow Combinators

The primary workflow combinators take one or more child components and deploy them as a workflow.

There are two main types of component.

- Single child components support a single child, usually called `action`. When it is running, the child normally takes on the name `running`.
- Some specific components support the `action` child and one or more specially named children. For example, the `Try` component supports the `normal` and `abnormal` children, to define the behaviour after the action `component` completes. Similarly, the `CounterAction` component has an `action` and a `counterAction` child.
- Other components support multiple children, in which case there are no naming policies. The `Sequence`, `Parallel`, and `Random` components are all such components. The order in which children are declared may or may not be significant -it depends upon the specific component.

Component	*Function*
`CounterAction`	A workflow that deploys its `action` on startup, and a separate `counterAction` action when terminating. It can make two workflow sequences appear to be a normal static deployment. To aid in this, the `liveness` child component is deployed -all `sfPing()` health tests are relayed to this child, which can implement the component's health logic.
`Delay`	Waits for a period and before deploying its sub-component.
`During`	Waits for a certain period of time after which it terminates its `action` if not already terminated – then terminates itself normally. Early termination of the `action` causes the parent to terminate with the same status message.

Component	Function
`Finally`	This component deploys its `action` child on startup, but only starts it when it is told to terminate. As a result, the child can be used to perform cleanup actions.
`Parallel`	Starts all sub-components in parallel. Terminates when all sub-components independently terminate. The components behaviour when it has no children is configurable; it can stay deployed (default), or it can terminate itself automatically.
`RandomSequence`	`RandomSequence` is a modified `Sequence` in which the child components are started in a random sequence.
`Repeat`	Executes its sub-component. If the sub-component terminates normally, repeats this a given number of times.
`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.
`Sequence`	Starts the sub-components one at a time, starting the next when the previous one terminates successfully.
`TestBlock`	A test specific workflow container that deploys an `action`. It will terminate this after a specific time, if it has not already terminated, and record whether or not the action terminated normally, abnormally or timed out.
`Timeout`	Waits 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.
`Try`	Tries to deploy its `action` sub-component. When this is terminated, the `Try` container will invoke the appropriate sub-component indexed by termination code. Its normal use is for rollback after failure.

4.2 Component Manipulation Components

In addition to the Workflow Combinators, there are other components provided for starting and stopping other applications as part of the workflow process:

Component	Function
`Attribute`	given a reference to an component, and the name of an attribute within it, add, replace or remove that attribute as appropriate.
`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
`Terminator`	given a reference to an application, call `sfTerminate()` upon it with an appropriate termination type, then terminate self.

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).

4.3 Event Processing Components

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

Component	*Function*
DoNothing	Does nothing for a period and terminates in a way defined by its description. Also provides event receipt and message printing, so is very useful for debugging and tracing of workflows
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.
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 separate sequences of actions.
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.
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.

5 Component use in a Workflow

Many SmartFrog components have been instrumented so that they can be used within a workflow, by being configurable to terminate or detach after performing their work.

Attribute	*Meaning*
sfShouldTerminate	should the component terminate itself (default false)
sfShouldTerminateQuietly	should the component terminate quietly (default false)
sfShouldDetach	should the component detach itself from the graph(default false)

Here is the policy implemented by components that read and act upon these attributes:

- If either sfShouldTerminate or sfShouldTerminateQuietly is true, the component terminates after performing its work
- If sfShouldTerminateQuietly is true, there is no notification of termination.
- If sfShouldDetach is true then the component is detached. This happens even if the component is not terminating at this time.
- If none of the attributes are set, or all are false then nothing happens.

To use any appropriately enabled component in a workflow, add

```
sfShouldTerminate true;
```

In the component description.

5.1 Implementing workflow support in a component

Since the workflow 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 only acceptable for short-lived components, and even then is complex. Re-entrancy rules prevent termination of the component from being directly initiated from inside the sfStart method, that is, sfTerminate may not be called in sfStart as it can create deadlock if RMI attempts to make re-entrant calls on separate threads.

Consequently, if termination is required at the end of the sfStart method, components should delegate to a

separate `TerminatorThread (org.smartfrog.common.TerminatorThread)` that can asynchronously detach and/or terminate nominated components.

To implement this workflow support to any component add

```
new ComponentHelper(this).sfShouldDetachOrTerminate(...)
```

at the end of `sfStart()` -or at the end of any thread that does startup work. This will trigger component detachment and/or termination if the appropriate attributes request it, and do nothing if they are not set.

We recommend that every component that does this is tested inside a workflow sequence with `sfShouldTerminate` set to true, a sequence that should complete within a bounded time. This verifies that the termination is initiated.

6 The Primary Workflow Combinators

Here are the workflow combinators; the components that can hold workflow components underneath.

6.1.1 CounterAction

This component is intended to aggregate two sequences into what appears to be a conventional "static" deployment. It deploys and starts its `action` child when starting, on startup, and deploys but does not start the `counterAction` child.

When the component is told to terminate, the `counterAction` is finally started, hopefully starting the component(s) underneath. We say hopefully, as it depends on how the system is being terminated. It may fail if the system is in deep trouble, and because trouble during termination is not reported upwards (merely logged), workflow users will have to examine the log to determine the results.

One other aspect of the component, is that if a `liveness` child component is declared, it is deployed (with the name `liveness`), from which point all all `sfPing()` health tests are relayed to this child, which can implement the component's health logic. Liveness probes are never relayed to the `action` and `counterAction` child components.

6.1.2 Delay

`Delay` starts a thread during its `sfStart` phase that delays a period of time, then deploys its `action` sub-component as a child with the name `running`.

```
Delay extends OptionalActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Delay";
    delaySchema extends Schema {
    //delay in milliseconds
       time extends Integer;
    }
    time 0;
}
```

`Delay` terminates when the child terminates, passing the termination record upstream. Here, for example, the normal termination is passed up with the message "end of sequence"

```
testSequence extends Delay {

      delay 200;

       action extends Fail {
        message "end of sequence";
        normal true;
       }

}
```

The component can also be used without a child, in which case it will remain deployed for the duration of the delay. Here is an example of a complex workflow that uses both delays; one inside a sequence and the other as

the activity that the Try component runs after a normal or abnormal termination.

```
t extends Try {
  action extends Sequence {

  -- extends CopyFiles;

  -- extends Delay {
   delay 5000;
  }

  -- extends Startup;

  }

  normal extends Delay {
    delay 2000;
    action extends Cleanup;
  }

  abnormal normal;

}
```

This example shows how composition of workflow containers can be used to express more complex workflows.

6.1.3 During

During runs its action child for a specified period of time and, if it has not terminated in that period, terminates it and itself normally.

```
During extends ActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.During";
    duringSchema extends Schema {
     time extends Integer {
       description  ##time in milliseconds to delay terminating the component.
        If less than zero, is set to MAX_INT.
        If zero, there is no delay #;
       }
    }

    time 0;
}
```

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. A time of -1 or less indicates "forever"

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

6.1.4 Finally

This component deploys its action child on startup, but does not start it until it itself is terminated.

As with CounterAction, there are no guarantees that this component is always run. It is better to use the Try component and specify the action to take when the child terminates, than to rely on being able to start a sequence when the entire SmartFrog graph is being shut down.

6.1.5 Parallel

Parallel deploys workflow components in parallel.

```
Parallel extends ActionsCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Parallel";
    parallelSchema extends Schema {
     asynchCreateChild extends OptionalBoolean;
     terminateOnEmptyDeploy extends OptionalBoolean {
       description ##flag which indicates that the parallel component
        should terminate immediately if there are no children to deploy#;
     }
     terminateOnAbnormalChildTermination extends OptionalBoolean;
```

```
    terminateOnAbnormalChildDeploy extends OptionalBoolean;
  }
  // Deploy children in parallel? Set to true to deploy every child component
  //in a new thread
  asynchCreateChild false;
  terminateOnEmptyDeploy false;
  terminateOnAbnormalChildTermination true;
  terminateOnAbnormalChildDeploy true;
}
```

`Parallel` is the closest workflow container to the standard `Compound`. It differs in a number of ways.

1. The sub-components are only created, initialised and started during the `sfStart` phase of the parallel. The components are deployed and started one after the other.
2. If `asynchCreateChild`, is `true`, then each child component is deployed and started in a separate thread. Failures to deploy a child can still be caught and propagated, but may surface after the initial parent has started successfully.
3. The `Parallel` component waits until each of the sub-components has terminated normally before terminating itself. By default, 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.

The component uses the three workflow attributes `sfShouldTerminate, sfShouldTerminateQuietly` and `sfShouldDetach`, to control how it acts when it is terminated. Three further attributes control what causes the component to terminate:

Attribute	***Meaning***
`terminateOnEmptyDeploy`	Terminate (normally) if there no children are listed in the deployment descriptor. If false (default), it remains deployed until externally terminated. This attribute is only used when `asynchCreateChild` is `true`; failure to deploy a child in a synchronous deployment will *always* terminate the deployment.
`terminateOnAbnormalChildTermination`	Propagate abnormal child terminations upwards by terminating Parallel if a child dies abnormally. This is `true` by default. Set to `false` to ignore the unexpected failure of children.
`terminateOnAbnormalChildDeploy`	Terminate the component if a child throws an exception while deploying. This can be more significant than the failure of a deployed child, which is why it is separated out. The default is `true`.

When components are deployed under Parallel with `asynchCreateChild` false, they are deployed with their lifecycles synchronised. All are created, then deployed, then started. Sibling components can therefore communicate with each other, knowing that by the time any component is started its peers will be in the deployed or started state.

There is no such synchronisation between components started on separate threads. There are no guarantees as to what state siblings will be in when a component is started.

Once successfully started, the child components are added to the parent graph and the private creation thread terminates itself. It is then the task of the parent to manage, ping and terminate the components. The worker threads are only used for startup.

To terminate child components in separate threads, regardless of how they were started, leave `sfSyncTerminate` at its default, `false`.

Here is an example parallel deployment of three virtual machines, in which each VM is started in its own thread. If any of the machines fails to deploy, the component terminates abnormally, but if a deployed VM terminates,

than the remaining machines are kept alive.

```
XenMachines extends Parallel {
  asynchCreateChild true;
  terminateOnAbnormalChildDeploy true
  terminateOnAbnormalChildTermination false;
  vm1 extends DeployVM { hostname "host1";}
  vm2 extends DeployVM { hostname "host2";}
  vm3 extends DeployVM { hostname "host3";}
}
```

A variation does not care if a child deployment fails; the number of virtual machines deployed can be between 0 and 2.

```
XenMachines extends Parallel {
  asynchCreateChild true;
  terminateOnAbnormalChildDeploy false
  terminateOnAbnormalChildTermination false;
  vm1 extends DeployVM { hostname "host1";}
  vm2 extends DeployVM { hostname "host2";}
}
```

System health in this use case has to be determined in some indirect means, perhaps by using a condition to probe services on the deployed machines via a `LivenessTest` component. For example, the system could be deemed deployed if a web page could be retrieved from a load-balancing front end. Consult the *Conditions* section later on in this document for more details on this approach.

6.1.6 RandomSequence

The `RandomSequence` component deploys each of its children once only, but chooses the order at random. The order is set by the seed value.

```
RandomSequence extends EventCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.RandomSequence";
    seed 6387315;
}
```

`RandomSequence` maintains a vector of the started sub-components, and deploys each child once, and once only. When started it deploys one component, and then whenever a child terminates, it starts a new one chosen at random. When the last terminates normally, it too terminates normally. If an error occurs at any point, or a sub-component terminates abnormally, the `RandomSequence` combinator terminates with an error.

The random number generator is the one behind `java.util.Random.`

6.1.7 Repeat

Repeated executes its child `action` repeatedly.

```
Repeat extends ActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Repeat";
    repeatSchema extends Schema {
       repeat extends Integer;
    }
    repeat 1; // set to -1 for infinite repeat
}
```

`Repeat` creates and starts a sub-component during its `sfStart` phase, as defined by the `action` template, and waits for its termination.

```
repeat extends Repeat {
  repeat 16;
  action extends WebApplication;
}
```

Whilst it is running, the child sub-component has the name `running`. If it terminates abnormally, so does `Repeat`. If it terminates normally, `Repeat` 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 repetitions is 1.

6.1.8 Retry

`Retry` starts its `action` child during its `sfStart` phase, then repeatedly redeploys copies of the action until a copy terminates normally. It can be used to handle operations that do not work every time.

```
Retry extends ActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Retry";
    retrySchema extends Schema {
       retry extends Integer {
        description "retry interval in milliseconds";
       }
    }
    retry 1;
    count 0;
}
```

If the child terminates normally, so does `Retry`. If the child terminates abnormally, `Retry` will start another instance of the `action` template. This continues until either the child terminates normally, or the deployment has been retried the specified number of times.

When the number of retries is exhausted, the workflow container terminates, forwarding the last status message up to the caller. In the example below, the component will be deployed eight times, until the final failure is accepted.

```
    example extends Retry {
      retries 7;
      action extends FailAbnormal {
        message "failure inside retry";
      }
    }
```

Every time that a new child is created, it is given a new name, "`running_`" plus the zero based count of retries. The first child is therefore `running_0`, the next `running_1`, etc. The `Retry:count` attribute will be set to the integer value of this count.

The minimum number of times the action is deployed is 1, even if `retry` is set to 0. Negative retries are not allowed.

6.1.9 Sequence

A `Sequence` takes each sub-component through its entire lifecycle, one at a time. It is the main building block of sequential actions.

```
Sequence extends ActionsCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Sequence";
    //control action on an empty deployment
    sfShouldTerminate true;
}
```

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.

An empty sequence is immediately self terminating, unless `sfShouldTerminate` is set to false.

`Sequence` is one of the most common of the workflow containers, because it can be used to execute any sequence of operations. Remember that it relies on the components terminating after their work is done. Many of the built in components can do this (a simple test is to check to see that they extend `WorkflowPrim`), however you must remember to set the `sfShouldTerminate` attribute to `true`. If a deployment appears to hang, this is often the cause.

Note that `Sequence` itself can be deployed in a `Sequence`, as can `Parallel` and the other Workflow containers.

The name of every deployed child is that of the name of the template component in the file.

6.1.10 TestBlock

This component is only mentioned for completeness; it is one of the components used to test SmartFrog components. It deploys its action and destroys it after a timeout. It notes whether the component was destroyed by a time out or was self terminating, and records this as attributes.

The component also implements a special remote interface, `TestBlock`, which provides remote access to the system state, and to the deployed `action` component.

6.1.11 Timeout

`Timeout` runs its `action` sub-component for a period of time and, if it has not terminated in that period, terminates it and itself abnormally. That is, it expects a its action to terminate successfully within a given period of time.

```
Timeout extends ActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Timeout";
    timeoutSchema extends Schema {
    //timeout in milliseconds
       time extends Integer;
    }
    time 0;
}
```

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.

The sub-component, when it is created, is known as `running`.

6.1.12 Try

`Try` creates and starts a sub-component during its `sfStart` phase, as defined by the `action` template, and waits for termination. It then runs another child depending upon how the child terminated.

```
Try extends ActionCompound {
    sfClass "org.smartfrog.sfcore.workflow.combinators.Try";

    /**
    * the name of the failing component is based on the
    * string returned from the fault. normal and abnormal
    * are the common ones, but there is nothing to prevent
    * different terminations from being used, even though
    * it is very unusual
    */
    trySchema extends Schema {
     normal extends OptionalCD;
     abnormal extends OptionalCD;
    }

}
```

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.

7 Event Processing Components

7.1.1 DoNothing

The DoNothing 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.

```
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 component prints the message to the log 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.

7.1.2 EventCounter

Event counter waist for events, when they arrive it decrements its counter (default 1). When the counter reaches 0, the component terminates.

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

If the attribute allDifferent is set to true, the counter is not decremented if the event it receives is an identical string.

7.1.3 EventQueue

The EventQueue component forwards events to registered components as all other workflow components do.

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

There are two 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.

7.1.4 EventSend

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

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

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

7.1.5 OnEvent

The `OnEvent` event dispatcher provides a number of named event handlers as attributes – defined as `LAZY` components.

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

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.

The component has two modes, defined by the `singleEvent` attribute. If this is `true`, only the first event that arrives is dispatched, and on termination of the associated action, the `OnEvent` component terminates. If the `singleEvent` attribute is `false`, the `onEvent` component dispatches an action for each event that it receives until it is terminated by another component or the dispatched event action.

If an event arrives for which there is no handler the event is simply ignored.

Once a handler has been started, it runs as the newly created sub-component as a child using the name of the event handler post-fixed by a unique index. When that child terminates, so does the parent `OnEvent` component if it is in single event mode, with the same termination type.

8 Component Manipulation Components

8.1.1 Attribute

Attribute replaces, removes or adds an attribute of a primitive or compound component.

```
AttributeSchema extends Schema {
    component extends CD;
    name extends String;
    // not present implies remove...
    value extends Optional;
}

Attribute extends EventPrim {
    sfClass "org.smartFrog.sfcore.workflow.components.Attribute";
}
```

The `component` is a reference to the component containing the attribute. 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 implies that the attribute should be removed.

8.1.2 Run

Run is a component that, during its start phase locates the action attribute – and deploys it as a separately running application. This application has a completely independent lifecycle from the workflow that creates it.

```
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
}
```

Once its action has been deployed, the run component terminates asynchronously. It terminates with an error if the application does not start correctly.

The launched application should be given a new name with the sfProcessComponentName attribute. The sfProcessHost attribute can set the host, while the sfProcessName attribute can be used to start it in a named daemon JVM.

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.

8.1.3 Terminator

Terminator is a component that, during its start phase, de-references its kill attribute to find a component. It then constructs a termination record from the type and description attributes and invokes the sfTerminate method on the referenced component. The termination type is set from the type attribute.

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

    terminatorSchema extends Schema {
       type extends String;
       selftype extends String;
       description extends String;
       detachFirst extends Boolean;
       kill extends OptionalCD;
    }
    type NORMAL;
    selftype NORMAL;
    //this is a constant
    NORMAL "normal";
    //this is a constant
    ABNORMAL "abnormal";
    description "terminator action";
    detachFirst false;
}
```

To terminate a remote process, the reference should 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.

After terminating the remote component, the terminator terminates itself, with the choice of termination code the selftype attribute. The description text is used in both the remote and local termination records. To avoid terminating a parent compound after the terminator terminates.

If the kill attribute is undefined, the terminator terminates itself but does not attempt to terminate any remote component. This can be used inside a workflow to signal to the container. As an example, the FailingWaitFor component is in fact a composition of WaitFor and Terminator:

```
FailingWaitFor extends WaitFor {
  message "Timed out";

  else extends Terminator {
    description PARENT:message;
    selftype Terminator:ABNORMAL;
  }
}
```

When deployed with an `action` component, this will terminate abnormally if the condition is not met within the specified time.

9 References

As the component tree 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. You cannot predict which components are deployed.

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.

In SmartFrog, when components are terminated, they are no longer exported as RMI objects, to enable garbage collection to take place reliably. Even if local references remain, they are not remotely visible. In a workflow, one is more likely to encounter terminated components within a component graph, so you need to be aware of this fact. Do not attempt to resolve references against terminated components, unless you wish to learn to recognise more RMI error messages.

10 Conditions

The *Condition* concept is an addition to the workflow components from late 2006. The core notion is simple: any SmartFrog component can be declared as a condition, by implementing the `org.smartfrog.sfcore.workflow.conditional.Condition` interface. These conditions can be used in conditional components that use the state of the condition to control their behaviour:

```
public interface Condition extends Remote {

    /**
    * Evaluate the condition.
    * @return true if it is successful, false if not
    * @throws RemoteException for network problems
    * @throws SmartFrogException for any other problem
    */
    boolean evaluate() throws RemoteException, SmartFrogException;
}
```

This interface's `evaluate()` method can be called to evaluate the current state of the condition, to see if it is true or false. These conditions can be used in *Conditional Components*, which are components that use the state of one or more conditions to alter their behaviour.

Conditional	***Function***
`If`	If/Then/Else semantics from live data. The `then` child is deployed if the condition is met, the `else` child otherwise.
`WaitFor`	Block until a condition is met. The `then` child is deployed if the condition is met, the `else` child if a timeout occurs.
`EvaluateCondition`	Evaluate a condition when the component is started; fail if it does not hold
`LivenessTest`	Test a condition on every liveness check; fail if the condition does not hold
`TestCompound`	The component to run tests against deployed components can use a condition to block the tests until the condition indicates the server is ready.

There are a number of conditions built into the SmartFrog core JAR

Condition	*Meaning*
`Value`	returns the current value of the `Value:value` attribute
`True extends Value`	returns true
`False extends Value`	returns false
`Equals`	tests that `left` equals `right` using the `Object.equals()` method.
`StringEquals`	tests that `left` equals `right` as case-sensitive strings
`Not`	evaluates `condition` and negates the result
`And`	Performs a Boolean "and" on all nested conditions. The test stops as soon as one condition returns false.
`Or`	Performs a Boolean "or" on all nested conditions. The test stops as soon as one condition returns true.
`Xor`	Performs a Boolean "xor" exclusive-or on all nested conditions. There is no way to shortcut this test.
`RunningFor`	Tests that the component has been deployed for a specific time, as specified in the `milliseconds`, `seconds`, `minutes`, `hours` and `days` attributes. This compound can be used inside `And` and `Or` conditions to help integrate delays into the testing process, expressing concepts such as "running for less than 1 minute or MySQL is live"
`IsPropertySet`	Returns true if the Java system property named by the string attribute `property` is set to a non-null value. The property is checked every time the condition is evaluated, so if it is set by some means, the result of this test can change.

There are some more complex conditions in specific packages. Some examples are as follows:

Condition	*Meaning*
`LivenessPage`	In the `sf-www` module. Can fetch a page and verify it was returned with the expected HTTP status code, the MIME type as expected, and with a response body it is above a specified size.
`FileExists`	In the `sfServices` JAR. Tests that a file exists, and can also check that it is above a minimum size.
`HostExistsCondition`	In the `sf-www` module. Checks that a named host can be resolved
`IsJdbcConnectionOpen`	In `sf-database`. Tests that a JDBC connection can be made.
`IsMysqlLive`	In `sf-database`. This condition that uses the `ping()` method on the JDBC driver to test for liveness.
`AssertCondition`	In the `sfServices` JAR. This is the `Assert` component set to only check any of its invariants during an `evaluate()` operation, returning false if one does not hold. This component/condition was written for to aid in test SmartFrog components.

Allowing such complex conditions to be written, while moving all policy on when to test the condition, and what to do on failure, is the goal behind the Conditions logic.

10.1 Writing your own condition

Any SmartFrog component can be turned into a condition. Condition components should not have any side effects; to not deploy anything or otherwise alter the state of the running system. These are meant to be tests for the state of system, not components that deploy parts of the system itself.

1. Extend the `Condition` interface.
2. Implement its `boolean evaluate()` method, return true when the condition is met.
3. Write some tests to verify the success and failure of the condition.

The `evaluate()` method is called whenever the container wishes to evaluate the result. It is only called after the component is already deployed and started.

Conditions that deploy subsidiary components (including nested conditions), can extend `org.smartfrog.sfcore.workflow.conditional.conditions.AbstractCompoundCondition`.

Here is a very simple example condition, the `BooleanValue` condition:

```
public class BooleanValue extends PrimImpl implements Condition {

    public static final String ATTR_VALUE="value";


    public BooleanValue() throws RemoteException {
    }

    public boolean evaluate() throws RemoteException, SmartFrogException {
       boolean value = sfResolve(ATTR_VALUE, false, true);
       return value;
    }
}
```

As you can see, basic conditions are very easy to write. This is their appeal: all policy that acts upon a the state of a condition is implemented in the containers, the conditional compound.

10.2 Writing your own conditional compound

To write a conditional component, subclass `org.smartfrog.sfcore.workflow.conditional.ConditionCompound`. This creates workflow combinator that by default, deploys its `condition` child at startup. Override the `isConditionRequired()` method to return false if this condition is optional, and override `deployConditionAtStartup()` if you want to delay the point at which the condition is deployed.

11 Examples

Here are some examples. Many of them are derived from the test applications

11.1 Empty Sequence

```
action extends Sequence {
 }
```

The simplest sequence is an empty one; nothing is deployed and the workflow continues

11.2 Simple sequence

11.3 Workflow with retry

Here is a Retry declaration, to retry a deployment

```
#include "/org/smartfrog/sfcore/workflow/combinators/retry.sf"
#include "/org/smartfrog/services/assertions/components.sf"
#include "/org/smartfrog/sfcore/workflow/combinators/sequence.sf"

action extends Retry {
       retries 7;
       action extends FailAbnormal {
         message "failure inside retry";
       }
     }
```

The nested action will fail every time it is started, so the retry component will try and deploy it seven times, before giving up.

11.4 Blocking until a state is reached

11.5 Multi-host application

There is an example of the use of the workflow system provided with the distribution – contained in package `org.smartfrog.examples.workflow`. This example is listed here.

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.

11.5.1 File base.sf:

```
#include "org/smartfrog/components.sf";
#include "org/smartfrog/functions.sf";
#include "org/smartfrog/sfcore/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"));
    #;
}
```

11.5.2 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";
        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 "jboss";
}
```

11.5.3 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;
}
```

11.5.4 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";

        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";
}
```