Guide to Orchestration in SmartFrog

Last Updated: 25/04/08

Contacts: andrew.farrell@hp.com, paul.murray@hp.com, patrick.goldsack@hp.com

VERSION 1.1

Note

This document gives details of what is in release 2.0 of the Smartfrog Orchestration engine. The engine is undergoing some final testing and minor enhancements. As a consequence, it is not currently checked into the open source repository. This will happen very soon. For now, it will be situated at: org.smartfrog.services.dependencies.experimental.*. This document has been made available now to give people some advance information of what's in the new release.

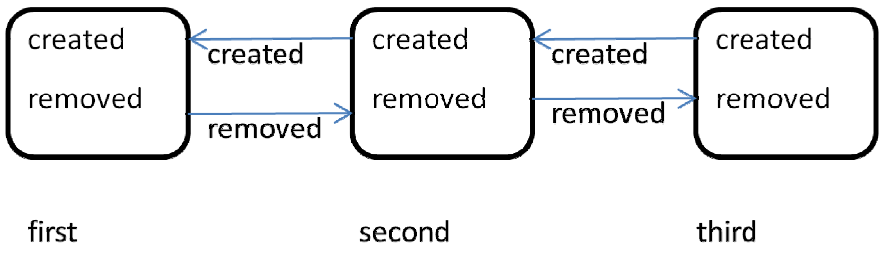
Introduction

The purpose of the orchestration extensions to SmartFrog is to provide a means by which the execution of logic within SmartFrog components may be orchestrated.

The fundamental concept is as follows. An orchestration is a set of SmartFrog components. Each component in the set defines dependencies which guard whether it may do some work and then change state.

As a trivial example, consider an orchestration consisting of three simple managed entities. Each managed entity may be created and subsequently removed. The initial state for each entity is that it is neither created nor removed. Consider also that the second (resp. third) may not be created until the first (resp. second) has been created. Conversely, the second (resp. first) may not be removed until the third (resp. second) has been removed.

This orchestration is depicted at least to some extent in the figure.



In the figure, there is for instance a dependency on the second managed entity (for it do something) that the first has been created. Not specified in the figure, but necessary for the example would be the fact that the dependency is only relevant if the second has not been created already.

Where to find stuff

The new release of the orchestration engine will be situated at:

org.smartfrog.services.dependencies.experimental

within the open source release of SmartFrog at www.smartfrog.org.

Example Orchestration Model

The following is a SmartFrog model with orchestration between components. It is a representation of the example orchestration that was presented in the Introduction.

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/dependencies/experimental/statemodel/components.sf"
#include "org/smartfrog/services/dependencies/threadpool/components.sf"

ManagedEntity extends StateComponent {
   sfClass "org.smartfrog.services.dependencies.experimental.examples.ManagedEntity";

   tcreated extends Transition {
      dependency dcreated;
      statefunction extends {
          created "true";
      }
   }

   tremoved extends Transition {
      dependency dremoved;
      statefunction extends {
          removed "true";
      }
   }

   created false;
   removed false;
   name "default";
}

ManagedEntities extends Model {

   foo0 extends ManagedEntity{
      name "foo0";
      dcreated "!created";
      dremoved "!removed && foo1:removed";
   }

   foo1 extends ManagedEntity{
      name "foo1";
      dcreated "!created && foo0:created";
      dremoved "!removed && foo2:removed";
   }

   foo2 extends ManagedEntity{
      name "foo2";
      dcreated "!created && foo1:created";
      dremoved "!removed && created";
   }
   sfTerminateCondition "foo0:removed";
}
```

Model

The model itself is specified as a prototype extending Model.

StateComponent

We define a component prototype called ManagedEntity which extends the StateComponent prototype defined in: "/org/smartfrog/services/dependencies/experimental/statemodel/components.sf".

```
ManagedEntity extends StateComponent {
   sfClass "org.smartfrog.services.dependencies.experimental.examples.ManagedEntity";

   tcreated extends Transition {
      dependency dcreated;
      statefunction extends {
          created "true";
      }
   }

   tremoved extends Transition {
      dependency dremoved;
      statefunction extends {
          removed "true";
      }
   }

   created false;
   removed false;
   name "default";
}
```

A state component will list a number of transitions, which are prototype descriptions which extend Transition. It will also list a number of attributes that compose its "working state". The working state of a state component is the sum of all of its attributes except its transitions. The working state is itself divided into two disjoint sets: orchestration and non-orchestration state.

The orchestration state of a component are those attributes whose values may only be changed in accordance with the transitions specified by the component. Non-orchestration state may be changed arbitrarily. Note that there is no tagging to indicate which is which – this information is inferred from the component description.

In the example, we define two transitions: one which manipulates the value of the "created" attribute, and another which manipulates the "removed" attribute. Any transition can manipulate any subset of the orchestration state. In this example, the orchestration state comprises "removed" and "created". The attribute "name" comprises the non-orchestration state.

Note that, for now, all components must be uniquely named.

Transition

A transition expresses two things: a dependency and a state function.

A dependency firstly is a string. It is represented as such for ease of manipulation in preparing models for input to a model checker. The format of the string is exactly Smartfrog syntax, except that one does not need to make operands LAZY – in being a string the dependency proposition will not get evaluated during link resolution.

It is a proposition based on the values of attributes (pertaining to orchestration state) within other components.

A state function expresses how attributes are to be updated. It is a map of attributes for update against a specification of how to update them. The specification is given as a string. It may represent some Boolean/arithmetic expression, a simple value literal, or a reference. It may also be a data description specifying alternative values for the attribute being set -- each attribute of the description specifying an alternative.

Examples of all of these possibilities are now given:

- Arithmetic/Boolean expression:

```
statefunction extends {
          foo “3*4”;
}
```

- Reference, here: bar

```
statefunction extends {
          foo "bar";
}
```

- Non-deterministic choice from alternatives. Here, foo is assigned to either 2 or 12.

```
statefunction extends {
          foo extends {
             -- “1*2”;
             -- “3*4”;
          }
}
```

In the example, tcreated’s dependency is given by the resolution of dcreated. For foo0, this is defined:

```
   foo0 extends ManagedEntity{
      name "foo0";
      dcreated "!created;
      dremoved "!removed && foo1:removed)";
   }
```

as being satisfied when foo0’s “created” attribute is false.

The state function for tcreated simply sets created to true, as can be seen in the previous presentation of the ManagedEntity prototype.

Component and Model Termination

A component or model may specify a value for sfTerminateCondition which specifies a condition which, when true, will entail the termination of the component/model. In the example model, the termination condition is that foo0's "removed" attribute is true.

Programming State Components

State components are programmed as extensions of
org.smartfrog.services.dependencies.statemodel.ThreadedState.

The ManagedEntity prototype specified in the orchestration model previously specifies the following component class in its sfClass attribute.

```
package org.smartfrog.services.dependencies.experimental.examples;

import java.rmi.RemoteException;

import
org.smartfrog.services.dependencies.experimental.statemodel.ThreadedStateComponent;
import org.smartfrog.sfcore.prim.Prim;

public class ManagedEntity extends ThreadedStateComponent implements Prim {

	public ManagedEntity() throws RemoteException {super();}

	public boolean requireThread(){ return true; }

	public boolean threadBody(){
		selectSingleAndGo();  //ignore return value...
		return true;
	}
}
```

Once a component is created and initialized (using the normal SF lifecycle), it will be invoked to do something whenever one of more of its transitions have their dependencies differently enabled.

An invocation consists of the following two steps:

- The requireThread method is called. The method returns "true" if the component determines that some action needs to be taken, "false" otherwise. If no action is taken, the component remains its in current orchestration state.

- If an action is required, this must be implemented in the threadBody method. The thread body will be placed on a thread pool queue for execution as soon as a thread becomes free. Note that a threadBody() also returns a value. This pertains to whether the action is to be made synchronously (i.e., prior to the threadBody() returning) or asynchronously (at some later time). We will come back to this point later.

If a further change happens before the threadBody is actually executed by the thread pool, it is removed and a new action placed on the pool. Any thread actually executing is allowed to complete, and commit/abort the transition.

There are a number of methods available within

org.smartfrog.services.dependencies.experimental.statemodel.StateComponent

which allow access and manipulation of orchestration and non-orchestration state from within requireThread and threadBody.

- `Context getLocalState()`, `Context getLocalOrchestrationState()`, and `Context getLocalNonOrchestrationState()` -- these methods make the local orchestration / non-orchestration state of the component available for inspection. Note that these are clones so changes made to the returned context will have no effect on component state.
- `HashMap<String, ComponentDescription> getPossibleTransitions() throws StateComponentSetAttributException` - this returns a hashmap of the currently enabled transitions for the component.
- `void setTransitionToCommit(String transition_name) throws StateComponentSetAttributException` – having studied the possible transitions returned by the previous method, one of them may be selected for initiation. If no such transition is available for selection, the given exception is thrown.
- `void setOrchestrationAttribute(String key, Object value) throws StateComponentSetAttributException` – used to set an attribute of orchestration state. Only may be used for attributes whose state function determines that there is a choice of values that may be assigned.
- `void setNonOrchestrationAttribute(String key, Object value) throws StateComponentSetAttributException` – used to set an attribute of non-orchestration state
- `void go() throws StateComponentSetAttributException` – commit changes specified in calling `setAttribute`
- `boolean selectSingleAndGo()` – This may be used when there is a single transition applicable to a component – will return false otherwise – with deterministic next values for attributes (according to the transition). That is, it does a select of transition (setTransitionToCommit) and go() all in one.

In the example, the dependencies are structured such that whenever either are satisfied for any of the three components, we will want to do something. Moreover, because in any particular instance there is only one transition that is applicable and its state function deterministically prescribes the next values of the pertinent attributes, we may simply call selectSingleAndGo() in order to effect the given transition. In all of the managed entities, the first applicable transition is to set “created” to true, the second is to set “removed” to true.

Continuing with the example, the threadBody completes its appropriate action by the time it returns. This is indicated by the fact that it returns true. This characteristic is fine unless a state component interacts with some external entity that can take a long time to complete – for example waiting for a server to boot which can take several minutes. This would simply block a thread from the thread pool and limits parallelism. For these cases, the threadBody can choose to complete asynchronously.

To complete asynchronously, the threadBody returns the value false to indicate non-completion. Through some internal mechanism or a call-back by some 3rd party code, an appropriate transition may be subsequently made to the component using the API methods presented above. Having returned false from threadBody, no further action is possible within the component whilst the asynchronous action is yet-to-complete.

There are times when completely asynchronous events occur – for example a part of the state that is managed by the state component fails, an internal timer fired, or some external management entity wants to force an action from the state component. In this case, the management (or other) logic must implement the callback interface:

org.smartfrog.services.dependencies.experimental.statemodel.InvokeAsynchronousStateChange.

This has a single method:

```
public void actOn(StateComponent sc) throws RemoteException;
```

The body of an implementation of this method constitutes the desired management logic. There is a similar to the one previously presented for StateComponent, allowing the selection of transitions, setting of attributes and the committal of changes.

There is an additional method in StateComponent, which when called causes the actOn method to be called:

```
public void invokeAsyncStateChange(InvokeAsynchronousStateChange iasc)
throws StateComponentSetAttributException
```

We pass the object implementing InvokeAsynchronousStateChange which has the desired implementation of management logic, as shown:

```
   public void managementMethod(…) {
       //whatever code

       statecomponent.invokeAsyncStateChange(new InvokeAsynchronousStateChange(){
              public void actOn(StateComponent sc) throws RemoteException {
                  HashMap<String,ComponentDescription> possible =
                                                sc.getPossibleTransitions(this);
                   …
              }
       }
   }
```

Thread Pool

For the orchestration engine to work it relies on a threadpool being available on each host. There are currently two SmartFrog configuration descriptions made available for thread pools. One uses the java.util.concurrent functionality available since Java 1.5, and the other uses our own implementation. There is not a great deal to choose between them for our purposes. The former description is located thus: org/smartfrog/services/dependencies/threadpool/simpletp.sf. The latter is located at: org/smartfrog/services/dependencies/threadpool/threadpool.sf.

Model Checking

An orchestration model is automatically checked for deadlock as a test of structural/behavioural integrity.

For example, consider the simple orchestration model:

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/dependencies/experimental/statemodel/components.sf"
#include "org/smartfrog/services/dependencies/experimental/examples/manents.sf"
#include "org/smartfrog/services/dependencies/threadpool/components.sf"


sfConfig extends Model {
   foo0 extends ManagedEntity{
      name "foo0";
      dcreated "foo1:created";
      tremoved;

   }

   foo1 extends ManagedEntity{
      name "foo1";
      dcreated "foo0:created";
      tremoved;
   }
}
```

In this model, it is evident that there is an inherent deadlock. (See previous definition of ManagedEntity.) That is, foo0's "created" attribute may not be changed to true until foo1's "created" attribute has been made true. But, the reverse is also true: "foo1:created" may not be made true until "foo0:created" has been made true. As there are no other transitions possible, the model is deadlocked.

Indeed, if we run this model through sfParse, it fails to parse with the following diagnostic output being produced:

```
c:\trunk\core\smartfrog\src\org\smartfrog\services\dependencies\experimental\exampl
es>sfparse deadlockdeadlock.sf
sfparse deadlockdeadlock.sf

Parser - SmartFrog 3.12.033dev (2008-04-26 01:24:35 BST)
```

```
(C) Copyright 1998-2008 Hewlett-Packard Development Company, LP

'deadlockdeadlock.sf':

SmartFrogResolutionException:: Deadlock detected in model:extends {
  <snip>
 with report:
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  foo1.created = 0
  foo1.sfTransitionInProgress = 0
  foo0.created = 0
  foo0.sfTransitionInProgress = 0
  deadlock = 1
  foo1.deadlock = 1
  foo1.sfTransitionDependency1 = 0
  foo1.sfComponentIsTerminated = 0
  foo0.deadlock = 1
  foo0.sfTransitionDependency1 = 0
  foo0.sfComponentIsTerminated = 0

	at
org.smartfrog.services.dependencies.experimental.modelcheck.ModelCheck.doit(ModelCh
eck.java:99)
	at org.smartfrog.sfcore.languages.sf.Phase.actOn(Phase.java:103)
	at
org.smartfrog.sfcore.componentdescription.ComponentDescriptionImpl.visit(ComponentD
escriptionImpl.java:842)
	at
org.smartfrog.sfcore.componentdescription.ComponentDescriptionImpl.visit(ComponentD
escriptionImpl.java:777)
	at
org.smartfrog.sfcore.languages.sf.sfcomponentdescription.SFComponentDescriptionImpl
.sfResolvePhases(SFComponentDescriptionImpl.java:916)
	at
org.smartfrog.sfcore.languages.sf.sfcomponentdescription.SFComponentDescriptionImpl
.sfResolvePhase(SFComponentDescriptionImpl.java:873)
	at org.smartfrog.SFParse.parseFile(SFParse.java:139)
	at org.smartfrog.SFParse.main(SFParse.java:297)
Error detected. Check report.
SFParse: FAILED
```

As can be seen, in the output from parsing the given orchestration model is diagnostic information directly from the model checker as to the nature of the deadlock. As it says, we are unable to advance from the initial state where “foo0:created” and “foo1:created” are both 0 (ie false).

It is also possible to test for arbitrary temporal constraints, written in CTL or LTL. See NuSMV documentation [NuSMV] for details of the syntaxes for these logics. These are attached as TLConstraint prototypes to Models/Components. For example:

```
foo extends Model {
     -- extends TLConstraint {
           constraint “AG (foo1:created -> foo0:created)”;
           result;
           details;
     }
}
```

The result of checking the `constraint` is written to `result` as true or false. If false, diagnostic details are written to `details` as a String. The details are truncated if too large.

Livelock: We allow livelock in a model, which is in keeping with the approach to orchestration presented here, under the criterion that sufficient dispensation is made within the model for all components/the model to terminate. This forces the model author to think properly about termination. That is to say, along all enactment paths there is always the possibility of termination.

Note that this condition, expressed as AG(EF terminate) in CTL, is weaker than saying that all enactment paths must eventually lead to termination: AF terminate. We enforce the former prescription but not the latter.

The model checking is done with the NuSMV model checker which has a liberal licence: LGPL. It will soon be checked into trunk\core\smartfrog\NuSMV. We currently only support the use of NuSMV under Windows, but soon we will also support Linux.

Formal Specification of Semantics

The formal specification of semantics is a working document, and can be found at:
http://smartfrog.svn.sourceforge.net/viewvc/*checkout*/smartfrog/trunk/core/smartfrog/docs/sfOrchestrationSemantics.pdf

Functional Tests

There is a single functional test – for the time being, located at:
org.smartfrog.test.system.dependencies.DependenciesTest.

References

[NuSMV] http://nusmv.irst.itc.it/