Guide to Orchestration in SmartFrog

Last Updated: 10/04/08

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

Note

This is an interim guide released prior to a significant update of the orchestration work. It is highly recommended to check back for updates. If you wish to be notified of the next release of the work, email andrew.farrell@hp.com. Note that the new work will be an evolution of the currently available release, so it is highly worthwhile understanding what is here already.

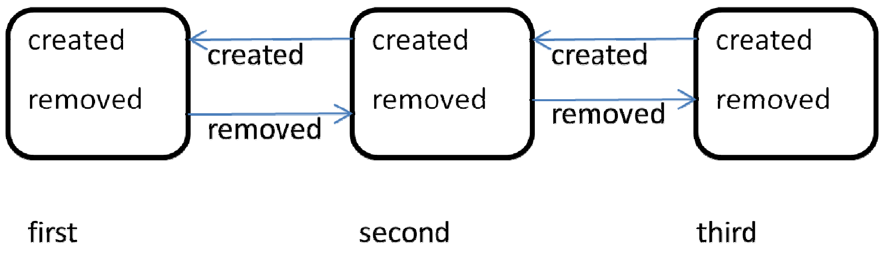
Introduction

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.

Notably, in the current approach we define dependencies for a component, but we do not define which states, or values for its attributes, a component may move to, or assume. This is really an omission and will be resolved in evolutions of this work. So, in the example, we should say that when first's created attribute is true and second is not yet created, second may just set its created attribute to true. Currently, the component will be asked to do something, but what is not specified. (In fact, it is asked whether it wants to do something in response to being enabled, and if so is allowed to do whatever it wants to do).

In order to achieve the desired functionality for this particular orchestration (as previously described), we must hard code certain aspects in the logic for the component. As said, in future a lot of what is hard coded will move into the orchestration model. In the following section, we present the model and the component logic which effects this orchestration along with a description of the modelling approach and implementation.

Where to find stuff

The orchestration engine is situated in org.smartfrog.services.dependencies 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/statemodel/components.sf"
#include "org/smartfrog/services/dependencies/threadpool/components.sf"

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

   [stateData, stateNotify] created false;
   [stateData, stateNotify] removed false;
   [stateData] sink false;
   [stateData] name "default";
}

createdDependency extends Dependency {
   enabled LAZY on:created;
   relevant (! LAZY by:created);
}

removedDependency extends Dependency {
   enabled LAZY on:removed;
   relevant LAZY by:created;
}

sfConfig extends Model {
   run true;

   foo0 extends ManagedEntity{
      [stateListen] -- LAZY foo1:removed;
      name "foo0";
   }

   foo1 extends ManagedEntity{
      [stateListen] -- LAZY foo2:removed;
      [stateListen] -- LAZY foo0:created;
      name "foo1";
   }
   foo2 extends ManagedEntity{
      [stateListen] -- LAZY foo1:created;
      sink true;
      name "foo2";
   }

   finished extends ModelTerminator {
      required LAZY foo0:removed;
      detachfirst true;  //remove
   }

   foo0Created extends createdDependency {
      on LAZY foo0;
      by LAZY foo1;
   }

   foo1Created extends createdDependency {
      on LAZY foo1;
      by LAZY foo2;
   }

   foo2Removed extends removedDependency {
      on LAZY foo2;
      by LAZY foo1;
   }

   foo1Removed extends removedDependency {
      on LAZY foo1;
      by LAZY foo0;
   }

}
```

State

We define a component prototype called ManagedEntity which extends the State prototype defined in “/org/smartfrog/services/dependencies/statemodel/components.sf”.

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

   [stateData, stateNotify] created false;
   [stateData, stateNotify] removed false;
   [stateData] sink false;
   [stateData] name "default";
}
```

As can be seen, certain attributes are variously tagged with “stateListen”, “stateData” and “stateNotify”. Attributes which are

- Tagged as “stateData” form part of the “consistent snapshot” of a component. Whenever a component is invoked to carry out some logic (as a consequence of being enabled according to its dependencies) these attributes will be provided to the component in a HashMap. Attributes outside this snapshot can be accessed using the normal SmartFrog attribute accessor methods (ie. sfResolve).

- Tagged as “stateListen” are those attributes whose values changing will cause a component to be invoked to do something. That is, a state component is only invoked when one if stateListen attributes (which will be references to other components’ attribute values) changes in value.

- Tagged as “stateNotify” are those attributes which cause the notification framework to force reevaluation of all component dependencies whenever they change value. For stateListen-tagged attributes to be pertinent, the attributes to which they refer must be tagged with stateNotify. An improvement of a future release is that this will be simply inferred, and the need to tag attributes with stateNotify will cease.

In the example model, changes to created and removed attributes in ManagedEntity state components are notified. Attributes created, removed, sink and name constitute the snapshot presented to such components. Elsewhere in the model, we extend ManagedEntity and prescribe some stateListen attributes for such extended components. For example:

```
foo1 extends ManagedEntity{
      [stateListen] -- LAZY foo2:removed;
      [stateListen] -- LAZY foo0:created;
      name "foo1";
   }
```

This prototype description specifies that foo1 is to be invoked to do something (subject to its extant dependencies being satisfied – see below) whenever foo2’s removed or foo0’s created attributes change value. This is specified by the tagging of the presented anonymous attributes with stateListen tags.

Tagging attributes within an orchestrated component correctly is essential to the correct execution of the model.

ManagedEntity extends a pre-defined component prototype State. State components are Prim-based components to be orchestrated. There is also a Composite component, which extends Compound, and must always be used instead for orchestrated state components.

Dependency

In the given model, we specify a number of dependencies between components. A dependency is defined to constrain processing of an orchestrated component by defining a pre-condition on that component. That precondition has a pair of predicates associated with it: relevant – when the pre-condition applies, and enabled – when the pre-condition is satisfied.

The general form for a dependency is:

```
myDependency extends Dependency {
    on LAZY someReference;     // to a state component or connector
    by LAZY another Reference; // to a state component or connector
    relevant aPredicate;
    enabled anotherPredicate;
}
```

The default for the two predicates if not supplied is true. A dependency is considered to be satisfied/enabled just in the events that relevant and enabled are both true or relevant is false (i.e. in propositional logic if relevant => enabled holds).

An example of a dependency from the presented orchestration model is:

```
foo0Created extends createdDependency {
      on LAZY foo0;
      by LAZY foo1;
   }
```

This is a dependency as can be seen between components foo1 and foo0. This prototype extends createdDependency which defines values for relevant and enabled:

```
createdDependency extends Dependency {
   enabled LAZY on:created;
   relevant (! LAZY by:created);
}
```

The values specified for these attributes means that the dependency is satisfied just in the events that foo0's created attribute is true and foo1's created attribute is false or foo1's created attribute is true.

The sum of dependencies specified for a component are implicitly subject to an AND-evaluation, meaning that for a component to be enabled all dependencies specified for it need to be satisfied for the component to be enabled.

ModelTerminator

Also in the model, we specify a ModelTerminator prototype, the name of which is finished. Whenever its required attribute evaluates to true, the model will be terminated.

From the example model:

```
   finished extends ModelTerminator {
      required LAZY foo0:removed;
   }
```

we see that the model will be terminated once the value of foo0's removed attribute becomes true.

Model

The model itself is specified as a prototype extending Model. This is an extension of SmartFrog's Compound prototype. Models define the attribute run. If set to true it will trigger the notification framework, whereby components are made aware of the progression of others. If set to false, the notification framework is disabled.

Connectors

There are connectors: AndConnector, OrConnector, NandConnector and NorConnector. They aggregate dependencies, according to the standard truth table semantics of AND, OR, NAND and NOR, respectively.

Dependencies which name a connector in their “by” attributes serve as the input dependencies for a connector. For an AND connector, if all such dependencies are satisfied at any time, then the connector is enabled (i.e. satisfied). For OR, just one such dependency needs to be satisfied. All dependencies which name a connector in their “on” attributes are enabled just when the connector is enabled and the evaluation of relevant=>enabled holds for the dependency.

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.examples;

import java.rmi.RemoteException;
import java.util.HashMap;

import org.smartfrog.services.dependencies.statemodel.state.ThreadedState;
import org.smartfrog.sfcore.common.Context;
import org.smartfrog.sfcore.prim.Prim;

public class ManagedEntity extends ThreadedState implements Prim {

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

	public boolean requireThread(HashMap data){
		return true;
	}

	public HashMap threadBody(HashMap data){

		HashMap save = new HashMap();

		boolean created = ((Boolean) data.get("created")).booleanValue();
		boolean sink = ((Boolean) data.get("sink")).booleanValue();
		String name = (String) data.get("name");

		if (!created) {
			System.out.println(name+" setting created to true");
			save.put("created", true);
		}

		if (created || sink) {
			System.out.println(name+" setting removed to true");
			save.put("removed", true);
		}

		//For test
```

```
            parent.put("output",cur_output);
            //

            return save;
      }

}
```

Once a component is created and initialized (using the normal SF lifecycle), it will be invoked to do something whenever its dependencies are enabled and its stateListen have changed. The attributes are considered "changed" when first deployed, so if the dependencies allow it, the component will be invoked at least once.

An invocation consists of the following two steps:

- The requireThread method is called; passing a map of the stateData attributes which is a snapshot of the component's orchestration attributes. The method returns "true" if some action needs to be taken, "false" otherwise.

- 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. The data is the same as for the requiredThread method even if the attributes have changed in the mean time.

- 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. This semantics is designed around the assumption that the attributes define the target state and that intermediate states can be ignored if they are superseded before they are achieved. Any thread actually executing is allowed to complete before a new action is placed on the pool.

In the example, the thread body will insert values for created and removed into the HashMap that is returned. It will set created to true, if its current value is false. It will set removed to true if its created attribute is already true.

The approach ensures that the modification of state attributes is synchronized appropriately with the snapshots used for the invocations described above. If one of the attributes modified within a component's threadBody (i.e. in the returned HashMap) is tagged with stateNotify, other state components are notified that they may need to re-evaluate their attributes and dependencies to see if the invocation steps are required.

In example above, the threadBody completes its appropriate action by the time it returns. This is indicated by the fact that the returned HashMap is not null. 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 do this the threadBody returns the value "null" to indicate non-completion. Through some internal mechanism or a call-back by some 3rd party code, a single call of the method "asyncResponseComplete " must be made passing the HashMap of attributes to be saved.

```
   public HashMap threadBody(HashMap data) {
      // whatever code is required (eg registering callback);
      // returning null to indicate an async completion
      return null;
   }

   public void callback(…) {
       //whatever code
       asyncResponseComplete(aHashMap);
   }
```

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 these cases, methods can be provided by the component to be called in these circumstances. Of course the full range of SmartFrog APIs are available for the implementation of these methods, however it is important to synchronize any changes to the external published state and the snapshot/notification mechanisms that are in use. Consequently, rather than use the sfReplaceAttribute method, and alternative method "saveState" is provided to enable several attributes to be replaced atomically with respect to this process.

```
   public void managementMethod(…) {
       //whatever code
       saveState(aHashMap);
   }
```

An alternative approach to asynchronous threadBody methods is to have the method return with an empty set of attributes, and for the completion to be considered as a completely asynchronous event that uses "saveState" to save the resultant attributes as opposed to the asyncResponseComplete method. The difference is that in the case of a null return and the use of the asyncResponseComplete, another action cannot occur until the response occurs. In the case of the empty HashMap and saveState other actions can occur as the thread management has been told that the action is complete.

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.

Example Output

```
foo0 setting created to true
foo1 setting created to true
foo2 setting created to true
foo2 setting removed to true
foo1 setting removed to true
foo0 setting removed to true
```

As one would expect from the foregoing narrative, we see from the print statements that each copy of ManagedEntity makes in its threadBody that foo0 first sets created to true, then foo1, then foo2. The sequence order is reversed for the removed attribute.

Formal Specification of Semantics

To be completed – see documentation for next release.

Simulation and Verification

To be completed – see documentation for next release.

Functional Tests

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