# Guide to Orchestration in SmartFrog

Last Updated: 08/04/08

Contacts: [andrew.farrell@hp.com](mailto:andrew.farrell@hp.com), [paul.murray@hp.com](mailto:paul.murray@hp.com), [patrick.goldsack@hp.com](mailto:patrick.goldsack@hp.com)

## Introduction

The purpose of the orchestration extensions 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 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 that the first has been created. Not specified in the figure, but necessary 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 clarified 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 set its created attribute 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 told 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.

## 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;

[stateListen, stateData] createdprev true;

[stateListen, stateData] removedsucc true;

[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{

name "foo0";

removedsucc LAZY foo1:removed;

}

foo1 extends ManagedEntity{

name "foo1";

createdprev LAZY foo0:created;

removedsucc LAZY foo2:removed;

}

foo2 extends ManagedEntity{

name "foo2";

createdprev LAZY foo1:created;

}

finished extends ModelTerminator {

required LAZY foo0:removed;

}

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;

}

}

We define a component prototype called ManagedEntity.

ManagedEntity extends State {

sfClass "org.smartfrog.services.dependencies.examples.ManagedEntity";

[stateData, stateNotify] created false;

[stateData, stateNotify] removed false;

[stateListen, stateData] createdprev true;

[stateListen, stateData] removedsucc true;

[stateData] name "default";

}

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

* Tagged as “stateData” form part of the “consistent snapshot” of a component. Whenever a component is deferred to 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 accesses using the normal SmartFrog attribute accessor methods (ie. sfResolve).
* Tagged as “stateListen” are those attributes whose values changing need to be notified to the component to act upon. Other attributes changing will not be notified.
* Taggeed as “stateNotify” are those attributes which, if changed through the “saveState” method (see below), will trigger notifications to other components who depend upon them.

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.

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 true.

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 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 foo0’s created attribute is false.

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.