Model-based Orchestration

andrew.farrell@hp.com

Last updated: 9th June 2008

Overview

Version 1.2 (of this documentation)

(best printed in colour!)

LABS hp Automated Infrastructure Lab SmartFrog

Contents

Change History

1.0 Initial tidy-up of original documentation `trunk/core/smartfrog/docs/SFOrchestration.pdf`

1.1 Initial documentation for new engine `trunk/core/smartfrog/docs/SFOrchestration1_1.pdf`

1.2 Release documentation for new engine `trunk/core/smartfrog/docs/SFOrchestration1_2.pdf`

This document = Version 1.2

1 Introduction

The purpose of the orchestration extension 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. For information regarding why this particular model of orchestration may prove beneficial over traditional workflow-based approaches to orchestration see [2].

A very simple example is the following. Let's say that there is a system of three managed entities, on which we wish to orchestrate management actions. 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 may not be `created` until the first has been `created`. Similarly, the third may not be `created` until the second has been `created`. Conversely, the second may not be `removed` until the third has been `removed`. Similarly, the first may not be `removed` until the second has been `removed`. This orchestration is captured in Figure 1.

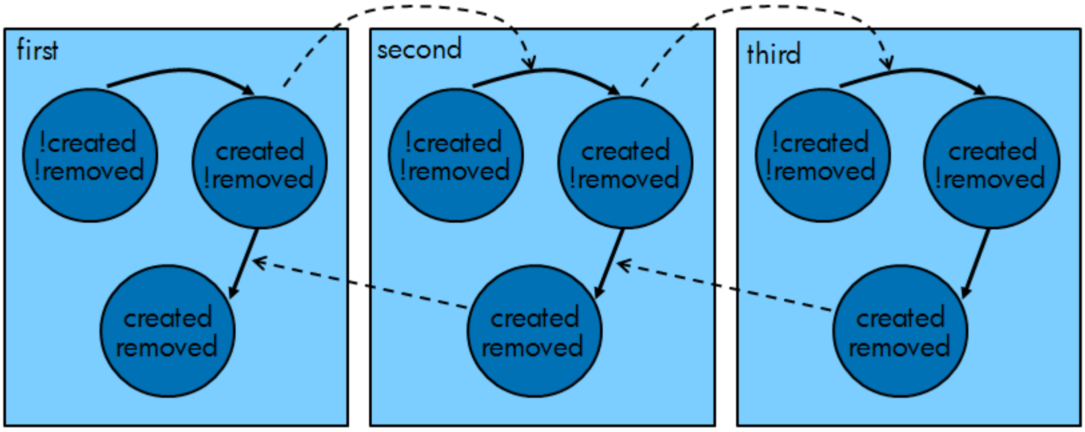

Figure 1: A Simple Orchestration

In the figure, there is for instance a dependency (dotted arrow) on `second` for it to move from `(!created,!removed)` to `(created,!removed)` states that `first`'s `created` attribute is `true`.

Where To Find Stuff

The new release of the orchestration engine is situated at:

`org.smartfrog.services.dependencies`

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

In order for the SmartFrog parser to perform verification of orchestration models, it is currently necessary to copy the appropriate `NuSMV` dynamic library from `trunk/core/extras/modelchecker/nusmv/lib` to `SFHOME/lib/NuSMV`. In time, this copy will be automated as part of the build process. Copy `NuSMVInterface.dll` if running SmartFrog under Win32 or `libNuSMVInterface.so` if under Linux32. Currently, we do not support verification of orchestrations for 64-bit operating systems.

2 Language Overview

Example Orchestration Model

```
#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";

   //Orchestration State
   created false;
   removed false;
   //Non-orchestration State
   name TBD;

   tcreated extends Transition {
      dependency (! LAZY created);
      statefunction extends {
          created true;
      }
   }
   tremoved extends Transition {
      dependency (LAZY created  && (! LAZY removed));
      statefunction extends {
          removed true;
      }
   }
}

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

ManagedEntities extends Model {

   killMe extends CompositeTerminator {
             terminateCond LAZY foo0:removed;
   }

   foo0 extends ManagedEntity{
      name "foo0";
   }
   foo1 extends ManagedEntity{
      name "foo1";
   }
   foo2 extends ManagedEntity{
      name "foo2";
   }

   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;
   }

   -- extends VerificationRecord {
             proposition "AG({foo0:created} -> {foo1:created})";
   }
}
```

Model

As may be seen from the foregoing example, an orchestration model is specified as (a prototype) extending `Model`.

State

In the example, 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";

   created false;
   removed false;
   name TBD;

   tcreated extends Transition {
      dependency (! LAZY created);
      statefunction extends {
          created true;
      }
   }

   tremoved extends Transition {
      dependency (LAZY created  && (! LAZY removed));
      statefunction extends {
          removed true;
      }
   }
}
```

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.

Transition

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

The dependency proposition specified within a transition is typically a dependency based on (a subset of) the orchestration state of the component specifying the given transition, although it need not be. Whether a transition is applicable at any time will depend in part on its dependency proposition as well as other factors – this is elaborated in *Formal Specification of Semantics*.

In the example, `tcreated`'s dependency proposition prescribes that the `created` attribute of the `ManagedEntity` be `false`.

A state function expresses how attributes are to be updated. It is a map of attributes for update against specifications of how to update them. A specification may be some simple value literal, a reference, or a data description specifying alternative values for the attribute being set – each attribute of the description specifying an alternative. A reference may only (directly or indirectly) refer to the *orchestration state* of the pertaining or other components.

Examples of all of these possibilities are now given:

- Simple literal value

```
statefunction extends {
   foo 2;
   foo2 12;
}
```

- Reference (including an APPLY reference, specifying an expression)

```
statefunction extends {
   foo LAZY bar;
   foo2 (LAZY bar1 + LAZY bar2);
}
```

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

```
statefunction extends {
   foo extends {
      -- 2;
      -- 12;
   }
}
```

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

External Dependencies

In a model, we may specify a number of dependencies *between* components, which we shall call *external* dependencies. These are in contrast to what we have described so far, namely dependencies *within* components – i.e. dependencies which guard when a transition is applicable for a component based on its current state.

An external dependency constrains the behaviour of a component by specifying a condition which must (at least) hold for the component to change orchestration state.

The condition has a pair of propositions associated with it: `relevant`, which determines when the condition applies, and `enabled` - which prescribes when the condition is satisfied if relevant.

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 aLazyProprosition;
    enabled anotherLazyProposition;
    transition ‘‘transition name’’;  //optional, could also be a vector of strings
}
```

A dependency is considered to be satisfied just in the events that enabled is 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 `foo0` and `foo1`. 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` 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.

An external dependency as described so far is a condition on the enablement of the state component as a whole. It can be made to be transition-specific through the inclusion of a `transition` attribute. This restricts the dependency condition to the named transition within the *by* component. Consider the following example.

```
dep extends Dependency {
    on LAZY zoo0;
    by LAZY zoo1;
    relevant LAZY on:bar0;
    enabled LAZY on:bar1;
    transition "fred"
}
```

Just transition `fred` within `zoo1` is subject to this dependency. If `transition` were `["fred", "bob"]`, then just “fred” and “bob” would be subject to the dependency.

Connector

There are a number of *connectors*: `AndConnector`, `OrConnector`, `NandConnector`, `NorConnector` and `XorConnector`. They aggregate dependencies according to the standard truth table semantics of AND, OR, NAND, NOR and XOR, 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.

Composite

Composites aggregate other composites, state components, dependencies and connectors, serving to promote modularity in the definition of a model. In this vein, they facilitate *compositionality* by exposing an interface by which a composite may be connected to other orchestration artefacts to form a model.

That is to say, the simplest way in which a fragment of one orchestration model may be reused in other or new models is for it to be bundled as a composite which enumerates a number of ports or connectors by which its behaviour may be influenced by the target model and vice-versa.

Input connectors are used to wire dependencies into the composite so that the behaviour of the composite is influenced by that of the rest of the model. Output connectors are used to wire dependencies out of the composite so that the behaviour of the rest of the model may be influenced by that of the composite.

From /org/smartfrog/services/dependencies/statemodel/components.sf, the definition of a composite is as follows.

```
Composite extends Compound {
     sfClass "org.smartfrog.services.dependencies.statemodel.state.Composite";
     sfUpdatable true;

     threadpool LAZY PROCESS:threadpool; // default place for the threadpool

     inputConnectors extends Compound {
        sfUpdatable true;
     }
     outputConnectors extends Compound {
        sfUpdatable true;
     }
}
```

Here, we see the input and output connector `Compounds`. Connectors within these `Compounds` are referred to by dependencies within/outside the composite. The general form of extending a composite is as follows.

```
myComposite extends Composite {
    inputConnectors:ic extends AndConnector;
    outputConnectors:oc extends AndConnector;

    s1 extends State { }
    s2 extends State { }
    c extends Composite { }

    d1 extends Dependency {
       on LAZY inputConnectors:ic;
       by LAZY s1;
    }
    d2 extends Dependency {
       on LAZY s2;
       by outputCOnnectors:oc;
    }

    etc
}
```

Note that a model is implemented as a `Composite` (with no input/output connectors).

CompositeTerminator

A composite may specify a `CompositeTerminator` which has the effect of terminating the composite when the `terminateCond` condition specified therein holds. The single instance in the previous model was:

```
killMe extends CompositeTerminator {
          terminateCond LAZY foo0:removed;
}
```

For the time being, a `CompositeTerminator` may only be attached to a model.

Verification Record

A verification record specifies a proposition in some temporal logic which must hold for the model when parsed. See *Verification* for more details.

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

public class ManagedEntity extends ThreadedState 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 as part of a notification cycle which fires whenever a transition is made within the orchestration model.

An invocation consists of the following two steps:

- The `requireThread` method is called. As its name suggests, the method is called to determine whether a thread is needed for the purpose of carrying out the work needed to make a transition. As an alternative, very light work may be carried out within the body of this method itself. Note carefully, however, as this method is called within the notification cycle it is important to ensure timely completion of its execution.

  The method returns `true` if the component determines that a separate thread is needed to make a transition, and `false` otherwise. It may also return `false` if no transition is to be made, in which case the component remains in its current orchestration state.

  `requireThread` may also be used to determine whether a transition which has previously been scheduled for a component, but which is yet to be carried out, should be removed. If it is not to be removed, then we would return `false`. If it should be removed, then we return `true`. It is not possible to remove a thread if it has been started already. In this case, the `requireThread` method would not get called in the first place, as the currently running transition would be allowed to complete.

- If a separate thread is required, we implement the logic to be run inside 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 transition should be considered to be complete when the `threadBody` returns (synchronous behaviour), or whether it will complete at a later time (asynchronous behaviour). We will come back to this point later.

There are a number of methods available within `org.smartfrog.services.dependencies.statemodel.State` which allow access and manipulation of orchestration and non-orchestration state from within `requireThread` and `threadBody`.

- `HashMap<String, Object> getLocalOrchestrationState()`,
  `HashMap<String, Object> getLocalNonOrchestrationState()`,
  `HashMap<String, Object> getLocalState()`,

  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`

  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 setAttribute(String key, Object value)`
  `throws StateComponentSetAttributException`

  Used to set an attribute of non/orchestration state. For orchestration state, it only may be used for attributes whose state function determines that there is a choice of values that may be assigned.

- `void go() throws StateComponentSetAttributException`

  Commit changes specified in calling `setAttribute`

- `boolean selectSingleAndGo()`

  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 a component is enabled, it will have a transition that it can make. 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.statemodel.InvokeAsynchronousStateChange`.

This has a single method: `public void actOn(State sc)`

The body of an implementation of this method constitutes the desired management logic. It may make full use of the API just presented in order to effect a transition for a component.

There is an additional method in `State`, 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

     comp.invokeAsyncStateChange(new InvokeAsynchronousStateChange(){
          public void actOn(State _comp) {
                 HashMap<String,ComponentDescription> possible =
                                              _comp.getPossibleTransitions();
                   ...
          }
     }
}
```

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`.

4 Formal Specification of Semantics

In this section, we introduce a formal account of the semantics of our orchestration approach. We divide the presentation into two parts. Firstly, we give a simplified account which enables the reader to get to the nub of the semantic characterisation. We then give an account which is much closer to the way in which the orchestration engine that we have implemented works, and which is much closer to what is exposed to a model author through the authoring front-end which has been implemented in this work.

The purpose of this second account is to give a precise crystallisation of the semantics of our modelling approach. The principal difference between the two accounts is that second account more readily facilitates compositionality of orchestration fragments. Interestingly, the second account can be shown to not add anything extra in expressivity, and any model expressed using the second account may be mapped to a model using the first account. Note that some model structure is lost in such a mapping meaning that, once mapped, it is not possible to perform a reverse mapping.

We start the presentation with some preliminaries before describing the two accounts of our orchestration approach.

Primitive Sets

We assume a number of *primitive sets*, which are pre-defined sets of primitive objects:

- `Strings` – the set of all string literals.
- `Boolean` – {`true`, `false`}.
- `Number` – the set of all numbers.

We also define `Object`, which is a universal set – the union of all primitive sets.

Non-primitive Sets

We also define the following non-primitive sets.

- $\mathcal{C}$ – the set of *components* used in a model.
- $\mathcal{P}$ – the set of *composites* used in a model.
- $\mathcal{D}$ – the set of all *dependencies* possible in a model. It is the smallest set for a model p containing:
  - `true` and `false`

- (t,k,v) if $c_t \in \mathcal{C} \ \wedge\ \exists m,T.c_t\texttt{=}(t,m,T) \ \wedge\ \exists v'.\texttt{map\_get}(m,k)\texttt{=}v' \ \wedge\ v \in$`Object` holds

  (Every key in each component may be used in a dependency with any value from `Object`)
- $(d' \ \texttt{\&} \ d'')$ if d' and d'' are also in $\mathcal{D}$
- $(d' \ | \ d'')$ if d' and d'' are also in $\mathcal{D}$
- $(d' \ \rightarrow \ d'')$ if d' and d'' are also in $\mathcal{D}$
- $(d' \ \leftrightarrow \ d'')$ if d' and d'' are also in $\mathcal{D}$
- $!d'$ if d' is also in $\mathcal{D}$

- $\mathcal{SF}$ – the set of all *state functions* possible in a model. Each state function given a *map* of (*key*,*value*) pairs will yield a new map whose set of keys is identical, but whose values may have changed. That is, for a map m and any state function sf:

  $\forall k.(\exists v.\texttt{map\_get}(m,k)\texttt{=}v \ \equiv \ \exists v'.\texttt{map\_get}(sf(m),k)\texttt{=}v')$

Auxiliary Functions and Predicates

In the following, we introduce model artefacts such as composites and components. There are a number of auxiliary functions and predicates that are used in this account.

- We make use of a *map* abstraction, which stores (key,value) pairs.

  $\texttt{map\_get}(m,k)\texttt{=}v$ – gets the value v for key k from map m

- For composites:

  $\texttt{composite}(n,p)\texttt{=}q$ extracts the composite q with name n from composite p

- For components:

  $\texttt{component}(n,p)\texttt{=}c$ extracts the component c with name n from composite p

- For dependencies:

  $\texttt{satisfied}(d,p)$ determines whether (internal) dependency d is satisfied in the context of model p

  $\texttt{satisfied}(d,t,c,p)$ determines whether transition t is enabled in component c according to internal dependency d against model p and any external dependencies that may exist in p

Account I Semantics

We present the first (i.e. simplified) account of the semantics for model-based orchestration. For convenience, we introduce the structure of a model as we go along rather than separately.

Model

A *model* is a triple: $(\mathcal{C},\mathcal{D},\mathcal{SF})$, that is, a set of components, a set of possible dependencies and a set of possible state functions. For simplicity, in the following we refer to $\mathcal{C}$,$\mathcal{D}$, and $\mathcal{SF}$ explicitly.

Models are ranged over by p,p',. . ..

Component

A *component* is a triple: (n,m,T), such that:

- n is a name (from `Object`) for the component which is unique wrt other components within the model
- m is a map, which stores (key,value) pairs
- $T \subseteq \mathcal{T}$ is a set of *state transitions*, where $\mathcal{T}$=`Object`$\times\mathcal{D}\times\mathcal{SF}\times$`Boolean`, where:
  - The `Object` value used in a state transition is a name for it that is unique within T of which it is a member.
  - The `Boolean` value used in state transitions indicates whether the transition has been initiated but not yet finished or committed (indicated by `true`), or not (indicated by `false`). Initially, this is always set to `false`.

Components are ranged over by c,c',. . . and sets thereof by C,C', ... Dependencies are ranged over by d,d',. . ., and sets thereof by D,D',. . .. State transitions are ranged over by t,t',. . . and sets thereof by T,T',. . .. State functions are ranged over by sf,sf',. . ..

The auxiliary function `component` is characterised thus. For a model p, comprising the set of components $\mathcal{C}$,

`component`$(n,p)=c \;\equiv\; \exists m,T\,.\,c\in\mathcal{C} \;\wedge\; c=(n,m,T)$

(Holds when c is a component of name n within model p.)

Step System for a Model

In the following, we characterise the behaviour of an orchestration model as a *step system*. This is similar in concept to a transition system, which is a common term, except that we reserve the term transition for state transitions. A state transition is a pair of steps – an initiating step where a particular state transition is chosen for a component but is yet-to-be-committed, and a committing/aborting step where the state transition may be either committed or aborted. If committed, the appropriate change in state occurs. The relation $\rightarrow^p$ defines individual steps (that is, initiating and committing/aborting steps).

The step system $\rightarrow^p$ entailed by a model p is the smallest set of pairs containing:

- $\texttt{(}p\texttt{,}p'\texttt{)}$ if the relation $\texttt{Step(}p\texttt{,}p'\texttt{)}$ holds, and p' is a *initial* model. (An initial model is a model which has been correctly initialised. This means that the `Boolean` value in state transitions, pertaining to whether the transition has been initiated, is universally set to `false` for all transitions.)

  (This characterises steps from the initial state.)

- $\texttt{(}p''\texttt{,}p'\texttt{)}$ if $\texttt{(}p\texttt{,}p''\texttt{)}$ is also in $\rightarrow^m$ and $\texttt{Step(}p''\texttt{,}p'\texttt{)}$ holds

  (This characterises steps from other states.)

where:

- $\texttt{Step(}p\texttt{,}p'\texttt{)} \equiv$

  $\exists n\texttt{.(}\exists c\texttt{,}c'\texttt{.component(}n\texttt{,}p\texttt{)=}c \wedge \texttt{StepComponent(}p\texttt{,}c\texttt{,}c'\texttt{)} \wedge \texttt{component(}n\texttt{,}p'\texttt{)} = c'\texttt{)}$

  (Step in one of the components)

  $\wedge$

  $\forall n'\texttt{,}c''\texttt{.}n' \neq n \supset \texttt{(component(}n'\texttt{,}p\texttt{)=}c'' \equiv \texttt{component(}n'\texttt{,}p'\texttt{)=}c''\texttt{))}$

  (No Expansion, and other components are identical)

- $\texttt{StepComponent(}p\texttt{,}c\texttt{,}c'\texttt{)} \equiv \texttt{StepCommit(}c\texttt{,}c'\texttt{)} \vee \texttt{StepInitiate(}p\texttt{,}c\texttt{,}c'\texttt{)}$

  (Either a committing or initiating step in a component.)

- $\texttt{StepCommit(}c\texttt{,}c'\texttt{)} \equiv$

  $\exists d\texttt{,}sf\texttt{,}n\texttt{,}n'\texttt{,}m\texttt{,}m'\texttt{,}T\texttt{,}T'\texttt{.}c\texttt{=(}n\texttt{,}m\texttt{,}T\texttt{)} \wedge c'\texttt{=(}n\texttt{,}m'\texttt{,}T'\texttt{)} \wedge \texttt{(}n'\texttt{,}d\texttt{,}sf\texttt{,true)}\in T \wedge$

  $\texttt{(}m'\texttt{=}sf\texttt{(}m\texttt{)} \vee m'\texttt{=}m\texttt{)}$

  (get new map by applying state function. We may abort the transition meaning that we offer the possibility of not making any changes)

  $\wedge$

  $\forall n''\texttt{,}d'\texttt{,}sf'\texttt{,}b\texttt{,}b'\texttt{.((}n''\texttt{,}d'\texttt{,}sf'\texttt{,}b\texttt{)} \in T \wedge b'\texttt{=false} \equiv \texttt{(}n''\texttt{,}d'\texttt{,}sf'\texttt{,}b'\texttt{) in } T'\texttt{)}$

  (get new set of transitions – all `false` regarding being initiating steps)

- $\texttt{StepInitiate}(p,c,c') \equiv$

  $\exists d,sf,n,n',m,T,T'.c=(n,m,T) \wedge c'=(n,m,T') \wedge (n',d,sf,\texttt{false})\in T \wedge$ $\texttt{satisfied}(d,p) \wedge$

  $\forall n'',d',sf',b,b'.$

  $((n'',d',sf',b')\in T' \supset (n'',d',sf,b)\in T \wedge (n'=n'' \wedge b'=\texttt{true} \vee n' \neq n''$ $\wedge b'=\texttt{false}) \wedge$

  $(n'',d',sf,b)\in T \supset (n'',d',sf',b')\in T' \wedge b=\texttt{false})$

  (For an initiating step, check dependency and set boolean indicating 'initiated' to true for particular state transition.)

The auxiliary predicate $\texttt{satisfied}(d,p$ is characterised as follows, for dependency d and model p.

$\texttt{satisfied}(d,p) \equiv d=\texttt{true} \vee$

(Satisfied when d is simply true or)

$\exists d'.d=!d' \wedge \neg\texttt{satisfied}(d',p) \vee$

(Satisfied when d is contingent on d' being false and it is so or)

$\exists d',d''.d=d'\&d'' \wedge \texttt{satisfied}(d',p) \wedge \texttt{satisfied}(d'',p) \vee$

(Satisfied when d is contingent on d' and d'' being true and they are so or)

$\exists d',d''.d=d'|d'' \wedge (\texttt{satisfied}(d',p) \vee \texttt{satisfied}(d'',p)) \vee$

(Satisfied when d is contingent on d' or d'' being true and one of them is so or)

$\exists d',d''.d=d' \rightarrow d'' \wedge (\neg\texttt{satisfied}(d',p) \vee \texttt{satisfied}(d'',p)) \vee$

(Satisfied when d is contingent on $!d'$ or d'' being true and one of them is so or)

$\exists d',d''.d=d' \leftrightarrow d'' \wedge (\texttt{satisfied}(d',p) \wedge \texttt{satisfied}(d'',p) \vee \neg\texttt{satisfied}(d',p)$ $\wedge \neg\texttt{satisfied}(d'',p)) \vee$

(Satisfied when d is contingent on d' and d'' both being true or both false and one is so or)

$\exists n,k,v,m,T.d=(t,k,v) \wedge \texttt{component}(t,p)=(t,m,T) \wedge \texttt{map\_get}(m,k)=v$

(Satisfied when d is a triple (t,k,v) and the key-value map for component with name t

in model p has a value of v for key k)

Account II Semantics

In the second account, we add a little more structure to the modelling language and its characterisation. We introduce the following further artefacts: composites, external dependencies and connectors.

Model

We now define a model to be a pair $(p, \mathcal{A})$ where:

- p is a composite (see below) which contains the orchestration model
- $\mathcal{A}$ is an aggregation of artefacts used in the orchestration model. It is a quintuple: $(\mathcal{C},\mathcal{P},\mathcal{E},\mathcal{D},\mathcal{SF})$ where:
  - $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{SF}$ are (resp.) the set of components used, the set of (possible) dependencies, and the set of (possible) state functions, in the model (as previously described)
  - $\mathcal{P}$ is the set of composites used in the model
  - $\mathcal{E}$ is the set of external dependencies used in the model

In essence, then, a *model* is a (single) composite, as shown in Figure 2. As will be described a composite itself contains a number of composites and components.

The model composite is now ranged over by p,p',....

For convenience, in the following narrative, we will refer to a model composite simply as a model, and refer to the constituents of $\mathcal{A}$ (namely, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{SF}$, $\mathcal{P}$, and $\mathcal{E}$), directly – i.e., without reference to their containing model.

Component

A *component* is now a quadruple: $(n,m,ac,\mathcal{T})$, where the additional member ac of the tuple is a `Boolean` value which determines whether the component may masquerade as an *and connector* for the purposes of dependency evaluation.

Composite

A *composite* is a quintuple: (n,P,C,IR,OR), where:

- n is a name (from `Object`) for the composite which is unique wrt the set of composites within a model.
- P is a set of composites.

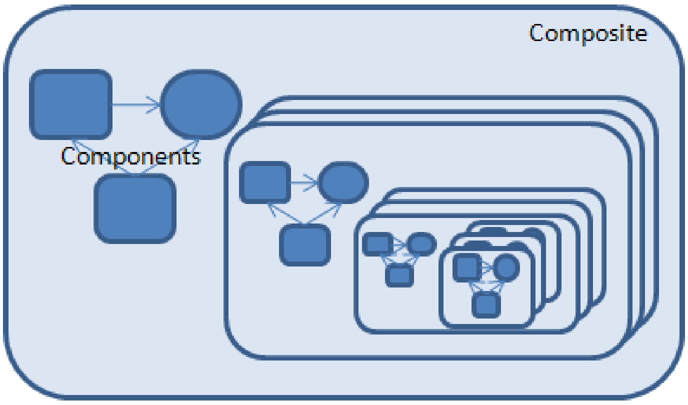

Figure 2: An Orchestration Model

- C is a set of components.
- IR is a set of input connectors.
- OR is a set of output connectors.

Composites are ranged over by p,p',...,q,q',.... Sets of composites are ranged over by P,P',...,Q,Q',....

The auxiliary function `composite` is thus characterised. For a model composite p with associated set of composites $\mathcal{P}$:

$\texttt{composite}(n,p)\texttt{=}q \equiv \exists P,C,IR,OR.q \in \mathcal{P} \wedge q\texttt{=}(n,P,C,IR,OR)$

(Holds when q is a composite of name n within model p.)

External Dependency

An *external dependency* is a sextuple: (n,on,by,r,e,TN), where:

- n is a unique name (from `Object`) for the dependency
- on is the name of an artefact (component or composite) within p
- by is the name of an artefact (component or composite) within p
- *relevant* is a dependency proposition (see previously) which may only refer to target attribute values which are part of the orchestration state within the on and by artefacts
- *enabled* is a dependency proposition with the same restriction as *relevant*
- TN is a set of transition names (can be empty) which constrain the applicability of the dependency to certain transitions of by (if it is a state component). If empty, the dependency is applicable to all transitions of by (if a state component).

Connector

A connector is simply one of *and*, *or*, *nand*, *nor*, and *xor*.

Step System for a Model

The definition of the step system for an Account II model is the same as that for Account I, with the exception of the definition of $\texttt{StepInitiate(}p\texttt{,}c\texttt{,}c'\texttt{)}$, which is now presented. Moreover, the only change is the replacement of (the use of) $\texttt{satisfied(}d\texttt{,}p\texttt{)}$ by (the use of) $\texttt{satisfied(}d\texttt{,}t\texttt{,}c\texttt{,}p\texttt{)}$ (where, for c read n and for t read n').

$\texttt{StepInitiate(}p\texttt{,}c\texttt{,}c'\texttt{)} \equiv$

$\exists d\texttt{,}sf\texttt{,}n\texttt{,}n'\texttt{,}m\texttt{,}T\texttt{,}T'\texttt{.}c\texttt{=(}n\texttt{,}m\texttt{,}T\texttt{)} \wedge c'\texttt{=(}n\texttt{,}m\texttt{,}T'\texttt{)} \wedge \texttt{(}n'\texttt{,}d\texttt{,}sf\texttt{,false)}\in T \wedge$

$\texttt{satisfied(}d\texttt{,}n'\texttt{,}n\texttt{,}p\texttt{)} \wedge$

$\forall n''\texttt{,}d'\texttt{,}sf'\texttt{,}b\texttt{,}b'\texttt{.}$

$\texttt{((}n''\texttt{,}d'\texttt{,}sf'\texttt{,}b'\texttt{)}\in T' \supset \texttt{(}n''\texttt{,}d'\texttt{,}sf\texttt{,}b\texttt{)}\in T \wedge \texttt{(}n'\texttt{=}n'' \wedge b'\texttt{=true} \vee n' \neq n'' \wedge$

$b'\texttt{=false)} \wedge$

$\texttt{(}n''\texttt{,}d'\texttt{,}sf\texttt{,}b\texttt{)}\in T \supset \texttt{(}n''\texttt{,}d'\texttt{,}sf'\texttt{,}b'\texttt{)}\in T' \wedge b\texttt{=false)}$

(For an initiating step, check dependency and set boolean indicating ‘initiated’ to true for particular state transition.)

The auxiliary predicate $\texttt{satisfied(}d\texttt{,}t\texttt{,}c\texttt{,}p\texttt{)}$ is characterised as follows, for dependency d, transition t, component c and model p.

$\texttt{satisfied(}d\texttt{,}t\texttt{,}c\texttt{,}p\texttt{)} \equiv \texttt{satisified(}d\texttt{,}p\texttt{)} \wedge \texttt{satisfied\_trans(}t\texttt{,}c\texttt{,}p\texttt{)}$

(A transition’s dependency is satisfied iff the internal dependency is satisfied and any external dependencies for the pertaining component are also.)

$\texttt{satisfied\_trans(}t\texttt{,}c\texttt{,}p\texttt{)} \equiv$

(A component’s external dependency is satisfied iff...)

(nb: the “trans” refers to the transition in question being taken into account – see note below)

$\texttt{satisfied\_trans\_noapp(}t\texttt{,}c\texttt{,}p\texttt{)} \vee$

(There is no applicable external dependency or)

$\forall ed \in \mathcal{E}\texttt{.}\exists o\texttt{,}r\texttt{,}e\texttt{,}T\texttt{.(}ed\texttt{=(}o\texttt{,}c\texttt{,}r\texttt{,}e\texttt{,}T\texttt{)} \wedge \texttt{(}T = \emptyset \vee t \in T\texttt{)} \supset \texttt{satisfied\_ed(}ed\texttt{,}p\texttt{))}$

(All applicable dependencies – i.e. those which name the transition in question, or whose set of pertinent transitions is empty – are satisfied.)

$\texttt{satisfied\_trans\_noapp(}t\texttt{,}c\texttt{,}p\texttt{)} \equiv$

$\forall ed \in \mathcal{E}\texttt{.}\exists o\texttt{,}r\texttt{,}e\texttt{,}T\texttt{.(}ed\texttt{=(}o\texttt{,}c\texttt{,}r\texttt{,}e\texttt{,}T\texttt{)} \supset \neg T = \emptyset \wedge \neg t \in T\texttt{)}$

(There is no applicable transition iff all external dependencies which name the component in their *by* attribute have a non-empty T – they name specific transitions which scope the dependencies – and the to-be-performed transition is not in any of the Ts.)

$\mathtt{satisfied\_ed}(ed,p) \equiv \exists o,b,r,e,T\,.\,ed=(o,b,r,e,T) \wedge$

(An external dependency is satisfied iff)

$((o \in \mathcal{C} \wedge \exists on,om,oac,oT\,.\,o=(on,om,oac,oT) \wedge$

$oac=\mathtt{true} \vee \neg o \in \mathcal{C}) \supset \mathtt{satisfied\_notrans}(o,p)) \wedge$

(If *on* is a connector or a component masquerading as an andConnector then it must be enabled/satisfied and)

$(\mathtt{satisfied}(r,p) \supset \mathtt{satisfied}(e,p))$

(relevant→enabled holds for the dependency)

$\mathtt{satisfied\_notrans}(c,p) \equiv$

(An *on* component/connector is enabled/satisfied iff...)

$\mathtt{satisfied\_notrans\_noapp}(c,p) \vee$

(There is no applicable external dependency or)

$(c \in \mathcal{C} \vee c=and) \wedge \mathtt{satisfied\_notrans\_and}(c,p) \vee$

(It is an andConnector or a component masquerading as one, and ... holds, or)

$c=or \wedge \mathtt{satisfied\_notrans\_or}(c,p) \vee$

(It is an orConnector and ... holds, or)

$c=nand \wedge \neg\mathtt{satisfied\_notrans\_and}(c,p) \vee$

(It is a nandConnector and the condition for an andConnector does not hold, or)

$c=nor \wedge \neg\mathtt{satisfied\_notrans\_or}(c,p) \vee$

(It is a norConnector and the condition for an orConnector does not hold, or)

$c=xor \wedge \exists ed \in \mathcal{E},o,r,e,T\,.$

(It is an xorConnector and)

$(ed=(o,c,r,e,T) \wedge T=\emptyset \wedge \mathtt{satisfied\_ed}(ed,p) \wedge$

(the condition for an orConnector holds, but...)

$\forall ed' \in \mathcal{E}.\exists o',r',e',T'(ed'=(o',c,r',e',T') \wedge T'=\emptyset \wedge \mathtt{satisfied\_ed}(ed,p) \supset ed=ed')$

(just one $T = \emptyset$ external dependencies may be satisfied)

$\mathtt{satisfied\_notrans\_noapp}(c,p) \equiv$

$\forall ed \in \mathcal{E}.\exists o,r,e,T\,.\,(ed=(o,c,r,e,T) \supset \neg T = \emptyset)$

(There is no applicable transition iff all external dependencies which name the component in their *by* attribute have a non-empty T)

$\texttt{satisfied\_notrans\_and}(c,p) \ \equiv$

$\quad \forall ed \in \mathcal{E}\,.\,\exists o,r,e,T\,.\,(ed\texttt{=}(o,c,r,e,T) \ \wedge \ T = \emptyset \ \supset \ \texttt{satisfied\_ed}(ed,p))$

(All $T = \emptyset$ external dependencies are satisfied.)

$\texttt{satisfied\_notrans\_or}(c,p) \ \equiv$

$\quad \exists ed \in \mathcal{E},o,r,e,T\,.\,(ed\texttt{=}(o,c,r,e,T) \ \wedge \ T\texttt{=}\emptyset \ \wedge \ \texttt{satisfied\_ed}(ed,p))$

(At least one of the component's $T = \emptyset$ external dependencies is satisfied.)

Note regarding `satisified_trans*` and `satisfied_notrans`:

When we assess the external enablement of a transition, each of the immediate dependencies for the transition may be (i) transition-specific, or (ii) non-transition-specific. We take into account both for immediate dependencies, hence the use of `satisified_trans*` predicates. Thereafter, when we "follow" dependencies, through their `on` specifiers, we only take into account non-transition-specific external dependencies, hence the use of `satisfied_notrans*` predicates.

Termination

In both semantic accounts, we may also specify a characterisation of termination. That is, for a particular model, if any condition of a `CompositeTerminator` for the model holds, then the model gets terminated.

To this end, we refine the definition of a model as follows. Note that, we present a characterisation of the second account of our orchestration semantics. The characterisation of the first account is almost identical.

We now define a model to be a triple (p, $\mathcal{A}$, $\mathcal{T}$) where the set $\mathcal{T}$ are termination conditions (from `CompositeTerminator`s associated with the model). A termination condition is a dependency from $\mathcal{D}$.

We update the definition of the step system for a model, by augmenting the definition of $\texttt{Step}(p,p')$. If we rename Account II's $\texttt{Step}(p,p')$ predicate to be $\texttt{Step}_{ACII}(p,p')$, then the new definition of $\texttt{Step}(p,p')$ is as follows.

$\texttt{Step}(p,p') \ \equiv \ \texttt{NotTerminated}(p) \ \wedge \ \texttt{Step}_{ACII}(p,p')$

(We may perform a step iff the model has not terminated and $\texttt{Step}_{ACII}(p,p')$ holds.)

The definition of `NotTerminated` for a model p is as follows.

$\texttt{NotTerminated}(p) \ \equiv \ \exists d \in T\,.\,\texttt{satisfied}(d,p)$

(There is a termination condition in $\mathcal{T}$ which is satisfied.)

5 Verification

As part of the parsing of an orchestration model, we perform a number of checks on the behavioural/structural integrity of the model. We check for:

- Model Deadlock
- Model Livelock
- Arbitrary temporal constraints on the model

We discuss each of these in turn.

Model deadlock is where the orchestration engine is not able to offer any component within the system any transitions that it may perform; and as a result the model is incapabable of being advanced in state. This is not a problem if the model has satisfied a termination condition in a `CompositeTerminator`, however. As a result, we characterise an absence of deadlock by the (Computational Tree Logic) `CTL` formula: `AG(deadlock -> terminated)`, which loosely says that if we have deadlock (no component can progress) then the model must be in a terminated state.

Model livelock *in a traditional sense* would be where there is a possibility of infinite looping in the enactment of a model; that is, there is a possibility that a model never converges to a termination state.

An absence of such a livelock would be characterised in `CTL` as `AF terminated`. This says that all paths must eventually terminate.

In this work, prescribing such an absence is far too restrictive. We want to allow the possibility of infinite behaviour. That said, we do consider it appropriate to prescribe that at any stage of enactment, it should be possible to terminate along *some path*.

Whereas `AF terminated` says that every path must finitely terminate, we want to allow the possibility of infinite paths but at every point along any path (i.e. after some finite number of steps along the path), it remains possible to terminate if so desired. This may be characterised as: `AG EF terminated`, which says that at any point of enactment, termination is brought about along some future path from the given point. This is the check that we make for livelock.

We also facilitate the checking of arbitrary `CTL` (Computational Tree Logic) and `LTL` (Linear Temporal Logic) temporal logic constraints against the model. This may be prescribed by the specification of `VerificationRecord`s at the top-level of a `Model` specification.

From `/org/smartfrog/services/dependencies/statemodel/components.sf`, the definition of a `VerificationRecord` is as follows.

```
VerificationRecord extends {
    proposition TBD;
    ltl false;
}
```

The `proposition` attribute specifies the LTL/CTL proposition to be checked, and `ltl` indicates whether the proposition is LTL (`true`) or CTL (`false`). The proposition is a string, where the syntax follows the LTL/CTL syntax specified in [1]. The atomic propositions that are present within a LTL/CTL proposition are delimited in curly braces. We see this in the example presented previously.

We have implemented an interface to the NuSMV model checker which checks the (absence of) deadlock condition of a model (`AG (deadlock` → `termination)`), the (absence of) livelock condition of a model (`AG EF termination`) and any arbitrary constraints expressed as `VerificationRecord`s.

We see the output of verifying the model presented previously for deadlock, livelock and the single constraint specified in the model. The proposition specified in the `VerificationRecord` is: `AG({foo0:created} -> {foo1:created})`, which says that whenever `foo0`'s `created` is `true`, `foo1`'s `created` attribute must be `true`. Clearly, in this example, this would not hold – `foo1`'s `created` attribute can not go to `true` until after `foo0`'s `created` attribute has.

As can be seen from the following output, the model checking process identifies the violated constraint and the parsing process fails.

```
C:\trunk\core\smartfrog>sfparse src\org\smartfrog\services\dependencies\examples
\meex.sf

Parser - SmartFrog 3.12.037dev (2008-06-02 23:16:59 BST)
(C) Copyright 1998-2008 Hewlett-Packard Development Company, LP

*0*DEADLOCK CHECK:PASSES
**********************************************
*1*LIVELOCK CHECK:PASSES
**********************************************
*2*AG({foo0:created} -> {foo1:created}):FAILS
***Verification record:
extends {
  sfIsVerificationRecord NULL;
  proposition "AG({foo0:created} -> {foo1:created})";
  ltl false;
  result false;
  failureRecord "C:/trunk/core/smartfrog/NuSMV/vrun2";
}
'src\org\smartfrog\services\dependencies\examples\meex.sf':
SmartFrogResolutionException:: Verification Run failure.  See foregoing output f
rom parse for details.

SmartFrogResolutionException:: Verification Run failure.  See foregoing output f
```

```
rom parse for details.
        at org.smartfrog.services.dependencies.modelcheck.ModelCheck.doit(ModelC
heck.java:114)
        at org.smartfrog.sfcore.languages.sf.Phase.actOn(Phase.java:103)
        at org.smartfrog.sfcore.componentdescription.ComponentDescriptionImpl.vi
sit(ComponentDescriptionImpl.java:842)
        at org.smartfrog.sfcore.componentdescription.ComponentDescriptionImpl.vi
sit(ComponentDescriptionImpl.java:777)
        at org.smartfrog.sfcore.languages.sf.sfcomponentdescription.SFComponentD
escriptionImpl.sfResolvePhases(SFComponentDescriptionImpl.java:916)
        at org.smartfrog.sfcore.languages.sf.sfcomponentdescription.SFComponentD
escriptionImpl.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
```

An output trace is provided in `C:/trunk/core/smartfrog/NuSMV/vrun2` as specified in the output from the trace (as given by the `failureRecord` attribute). This is output directly generated by `NuSMV`. It is not the simplest of reads and an abstraction of it would need to be provided as feedback to a model author.

It is worth mentioning that we do *not* regard the question of verification performance to be an important issue. `NuSMV` has a capability of handling 10-15 orders of magnitude of state variables. It is unlikely that any deployed orchestration will reach this level of complexity either in itself or without being capable of being broken down into logically-independent sub-orchestrations which may be verified separately.

6 Requirements of Authoring Tool

We make three observations regarding requirements for an authoring tool. These concern: initiation of verification, desired state representation and representation of state component transitions.

Initiation of Verification

An authoring tool for the orchestration engine should expose the option for the model author to perform design-time verification of the current model at any time, e.g. initiated with a "check model" button. The results of the verification process should be presented in a meaningful and understandable way.

Desired State Representation

The authoring interface should make explicit the notion of target state at the composite (including model) level. This just maps to another component, whose state may be changed by asynchronous management logic. This is a sufficiently useful concept to make it distinct, however. Other components in the composite will make transitions according to the current desired state attached to the composite. An example of the explicit modelling of desired state is shown in Figure 3.

In the figure, all of the transitions have "coloured-dot" dependencies on them. These indicate necessary conditions related to "current desired state" for their enablement. (Some have further conditions too.) For any of the transitions, one of the coloured desired states must hold for the transition to be enabled. For instance, for `OS`, the desired state for the composite/model must be `prepared` or `installed` for the transition from `!installed` to `installed` in `OS` to be enabled. This transition has a further condition, namely that `VM`'s `created` attribute is set to `true`.

Note also, for interest, that there is a termination condition on the model which holds when the desired state is `removed` and `VM`'s `created` attribute is `false`.

Representation of State Component Transitions

Note that for all but the simplest component descriptions, it will *not* be feasible to represent each state of a component individually. This is because there are potentially significantly many states.

As a result, what should primarily represented are *transition schemas* between a source *set* of states and a target *set* of states. Refer to Figure 4.

The primary currency for specifying the behaviour of components is thus individual transition schemas, rather than specifying a state machine as a single unit. Notice from

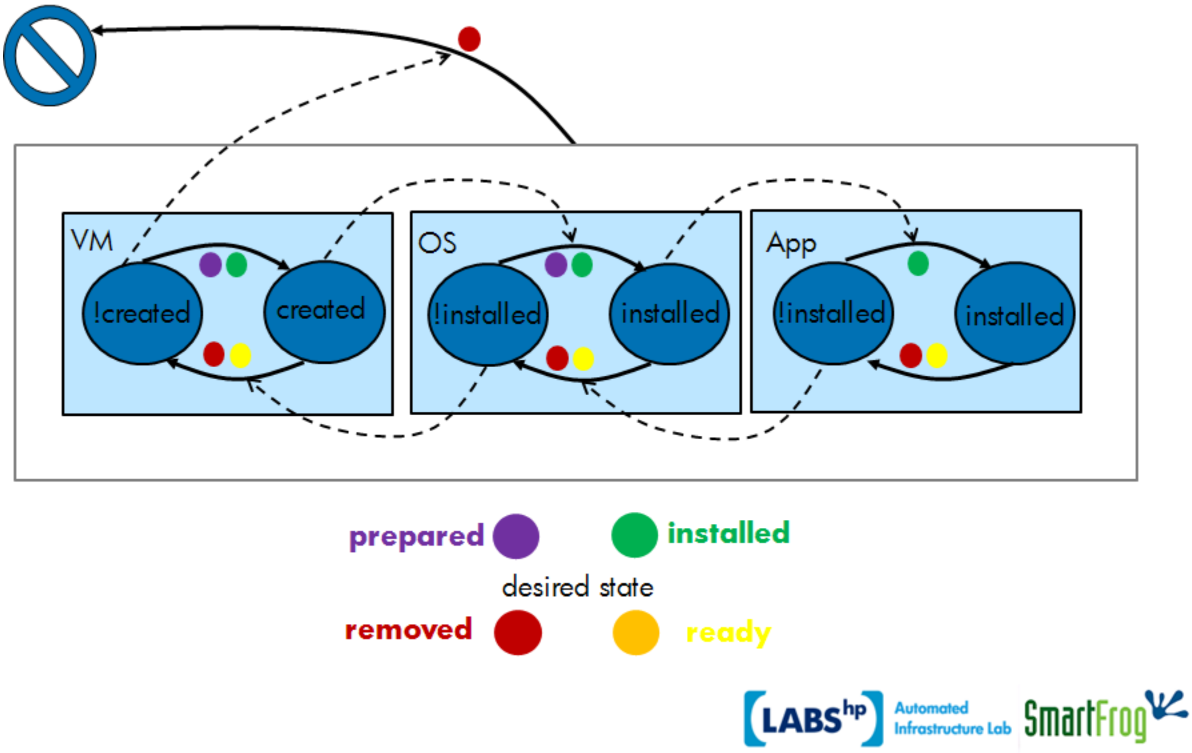

Figure 3: An Orchestration with Explicit Desired State

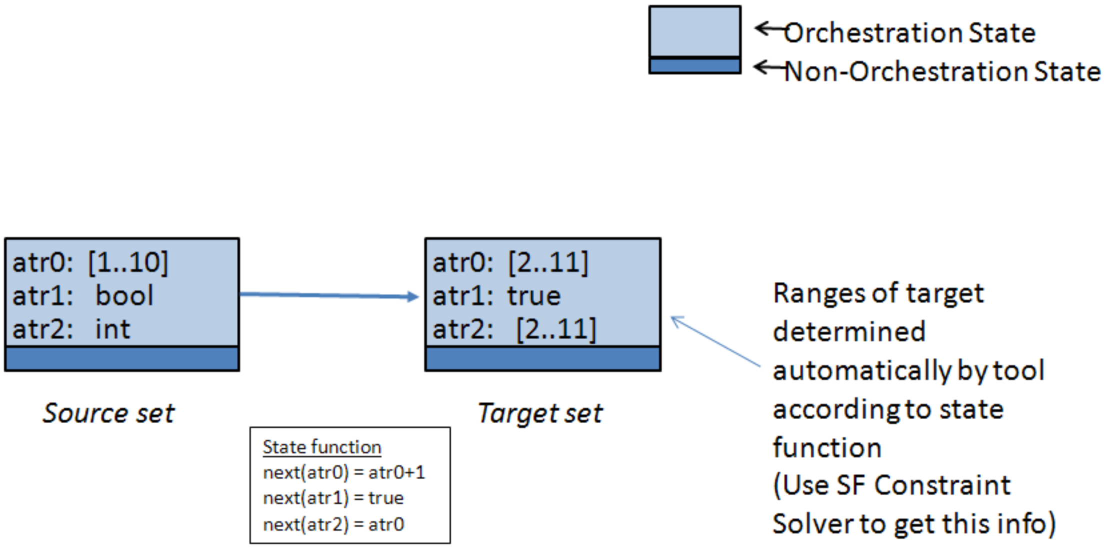

Figure 4: A Single Transition Schema within a State Component

the figure that the source of a transition schema is a set of states – it does fix precise values for the attributes, which means that the source is not a representation of a single state but rather a set of them. This holds similarly for the sink/target set of states.

It should be possible to specify the state functions of a transition schema with the authoring tool, and for the ranges of the attributes in the target set to be derived *auto-*

matically by the tool. The tool can use the SmartFrog Constraint Solver, for instance, to derive the new ranges of attributes.

It should be feasible for meaningful names to be given to source and target state sets which may aid readability of the model. Moreover, when the tool recognises that the source set of one transition schema is a subset $\subseteq$ of the sink/target set of another, it may link them visually. Finally, in degenerative/simple cases, the source and target may correspond to a single state, and if this is the case throughout the model it will be possible to represent the transition schemas (which are now just plain transitions) as a single state machine.

The `created`/`removed` `ManagedEntity` example which was discussed at the start of this document is an example of a degenerative case where transition schemas are transitions and where the authoring tool should link the transitions together into a single state machine.

7 Run-time monitoring

Run-time monitoring should be enabled, *for example* to:

- Provide feedback of current state
- Allow authors/users to turn certain artefacts off/on, such as dependencies or components themselves, say
- Allow run-time model checking of arbitrary constraints, e.g. so that invariants may be checked (which may not have been guaranteed in the first place)
- Identify when the current orchestration is deadlocked, save for changes that may be made by asynchronous management logic.

  When we check for deadlock in verification, there is still the possibility that no component at some stage of enactment wants to make a transition. All the deadlock check that we perform at design-time verifies is that some component can always make a transition. There is a difference here between *can* and *wants*. The design-time check covers the *can*, a run-time check should cover the *wants*.

8 Local Orchestration -v- Scalable Orchestration

The current focus of this work sits along the whole spectrum between the extremes of small, very localised orchestrations whose actions are tightly synchronised and large-scale, fault-prone, distributed orchestrations. For the second of these, it is imperative that components be endowed with sufficient autonomy so that they may perform actions when they deem them appropriate. This dispensation is at odds with the first enumerated extreme of orchestration, where the performance of actions is prescribed for components as part of some globally synchronised behaviour, where they are forced to be in lock-step with each other. This highly-synchronised form of orchestration will simply not scale, and is not tolerant to faults such as components disappearing.

For the time being, at least, we are interested in investigating how we might usefully support applications of orchestration at arbitrary points along the described spectrum. As a result, we keep an open mind regarding the types of modelling artefacts that we should support in our orchestration language. It is notable that the current modelling language is very much oriented towards the scalable extreme, but even for this purpose there is much work that could be done in fine tuning our approach. Our principle interest here is in management orchestrations where there may be potentially tens or hundreds of thousands of components that need managing. Imposing tightly synchronised behaviour in this context is simply not feasible. In contrast, we also have applications of orchestration that we are interested in supporting that sit towards the localised extreme.

9 Functional Tests

There are a number of functional tests located at: `org.smartfrog.test.system.dependencies.*`

References

[1] Edmund M. Clarke, Jr. and Orna Grumberg and Doron A. Peled. *Model Checking, ISBN: 0-262-03270-8.* MIT Press, Cambridge, MA, USA, 1999.

[2] Patrick Goldsack and Paul Murray and Andrew Farrell and Peter Toft. Smartfrog and Data Centre Automation. *The Rise and Rise of the Declarative Datacentre (R2D2), Microsoft/HP Joint Workshop, May 2008, Cambridge, United Kingdom.*