7 Published State Interfaces

All state in the previous operational signatures was opaque to the Consumer (eg. either as `navigationalState` or a `sessionID`). In addition, a Producer MAY expose information about entitity-specific state to a Consumer by publishing a model of the entitity's **Properties**. We introduce notation to define property models following the XFORMS specification, and then show how subsets of this information may be retrieved using the getPropertyDescription operation (the type definitions, access level, and scoping), and the getProperties/setProperties operations (the instance values of the properties themselves).

7.1 Property Models

A Property model includes type definitions in the form of a schema, instance values for each property, and property type bindings, which indicate the data type and access level The proposed XFORMS-based property notation form is as follows:

```
<model id="MyEntityProperties" Scope="Entity|Session">
      <xforms:schema>
          <!—type definition schema (optional)-->
      </xforms:schema>
      <xforms:instance>
          <!—current values of property elements – initialization data for the Entity-->
      </xforms:instance>
      <xforms :bind …/>* <!—property type, access level, optionality declarations-->
</model>
```

In sum, the design is to treat WSRP/WSIA properties, both at Entity and Session scopes, as elements in XFORMS models, and to reuse the existing XFORMS specification to describe their type, structure, and exchange instance values rather than proposing a new notation for WSRP/WSIA. Note that both "flat" and "hierarchical" models are possible under full control of the Producer simply by how it chooses to expose its model (i.e., write the model schema) – as a set of top level elements only or as a hierarchical model.

As a running example throughout this section we adopt the following scenario, in which a Consumer is integrating a configurable portlet provided by a Producer. Let us assume that the portlet in question displays the most recent press releases published to a newswire service for a specific company. The portlet is being integrated into a corporate intranet portal by an administrator, who wishes to configure the company name parameter (exposed state) of the portlet such that portal users see their company's press releases. In this case the full model notation would appear as follows (assuming the portlet is configured to display OASIS press releases):

```
<model id="PressReleaseEntity" scope="entity">
      <xforms:instance>
          <companyName>OASIS</companyName>
      </xforms:instance>
      <xforms :bind ref="companyName" type="xsd :string"/>
</model>
```

In the above example, no schema component of the model is required because the single property uses a built-in data type.

7.2 Property Operations

```
propertyDescription = getPropertyDescription(consumerContext, handle);
```

Where:

`consumerContext` is an extensible data structure defined in Section 11 with a set of references the Producer MAY use for locating the current state of an entity.

`handle` provides a context the Producer MAY use to determine which set of properties are to be described.

`propertyDescription` is a subset of the full model, omitting the <instance> values.

For example:

```
<model id="PressReleaseEntity" scope="entity">
      <xforms :bind ref="companyName" type="xsd :string"/>
</model>
```

```
propertyInstance = getProperties(consumerContext, entityContext, name[]);
```

Where:

`consumerContext` is an extensible data structure defined in Section 11 with a set of references the Producer MAY use for locating the current state of an entity.

`entityContext` is an extensible data structure, defined in Section 11, which includes a reference to the entity for which the Consumer is requesting current property settings. It also contains that entity's state in the case where the entity chooses to push this state to the Consumer.

`name[]` is the array of property names the Consumer is requesting. A null array MUST be treated as a request to enumerate the current published state of the entity.

`propertyInstance` is a subset of the full model, containing just the <instance> values.

For example:

```
<model id="PressReleaseEntity" scope="entity">
      <xforms:instance>
          <companyName>OASIS</companyName>
      </xforms:instance>
</model>
```

```
propertyInstance = setProperties(consumerContext, entityContext);
```

This signature is probably broken in the current draft, and needs to return an entityContext or interactioncontext depending on scope (see note below in "issues" section) rather than simply a property array.

Where:

`consumerContext` is an extensible data structure defined in Section 11 with a set of references the Producer MAY use for locating the current state of an entity. This includes a `sendPublishedState` flag indicating whether or not the Consumer is interested in receiving back the current published state of the entity as a property array.

`entityContext` is an extensible data structure, defined in Section 11, which includes a reference to the entity the Consumer is setting properties on as well as the properties being set. It also contains that entity's state in the case where the entity chooses to push this state to the Consumer. The instance component of the property model (as described above) would be carried inside the entityContext,.

`propertyInstance` is a subset of the full model, containing just the <instance> values.. If the `sendPublishedState` flag in the `consumerContext` parameter is 'true', the entity will return these in this property array.

7.3 Design-Time Interaction Sequence

Continuing with the press release portlet example introduced above, we assume that the Consumer portal provides a generic configuration tool which may be used by the administrator to set the value of any portlet property. The getPropertyDescription() method is first called by the portal to describe the properties published by the portlet and presented to the administrator for editing (for example, by dynamically constructing a form containing the properties as table rows). The getProperties() method is then called by the portal to obtain the current values of the properties for defaulting the form fields when initially presented to the administrator. setProperties is used for updating the properties once the form is submitted and retrieving any side-effects the Entity may make on related properties that may change as a result of the explicit changes made by the Consumer:

1. registerConsumer(portal) – the portal establishes a connection with the Producer [only on the first connection]
2. cloneEntity(press release portlet) – the portal clones the press release portlet for configuration [assume Consumer somehow finds the original entity]
3. getPropertyDescription(press release portlet clone) – the portal queries the press release portlet clone for the description of configurable properties
4. getProperties(press release portlet clone) – the portal queries the press release portlet clone for configurable properties' default values [steps 3&4 could be batched into a single round-trip]
5. setProperties(press release portlet clone, company name/value) – the portal sets the value of company property [returns new entity context reflecting new state]

7.4 Combining Session And Entity Scoped Models

Once we have the above XFORMS-based notation for properties, we can in the same model document return information about multiple models managed by the component – at either entity or session scopes. Thus we can partition entity-scoped properties into separate models where desired, perhaps to support Registration-based differentiation of the properties published by an Entity, or to scope the properties based on the "page" being displayed by the Entity.

We can also return multiple model descriptions when they are defined at different scopes, i.e. Entity and Session. An example property description that combines models at these two scopes would be encoded as follows:

```
<wsrp>
      <model id="PressEntityProperties" Scope="Entity">
          <schema xlink:href="Schema-PressEntity.xsd"/>
          <instance>
              <!—current values of properties– initialization data for the Entity-->
          </instance>
      </model>
      <model id="PressSessionProperties" Scope="Session">
          <schema xlink:href="Schema-PressSession.xsd"/>
          <instance>
              <!—current values of properties– initialization data for the Session-->
          </instance>
      </model>
</wsrp>
```

7.5 Example: Interdependent State

We describe another, more complex example in which an administrator is configuring a graphical portlet containing width and height properties in which the aspect ratio is constant; either the width or height may be modified and the other is automatically updated by the Producer. In this example the administrator sets the width and the height is automatically updated.

1. registerConsumer(portal) – the portal establishes a connection with the Producer [only on the first connection]
2. cloneEntity(portlet) – the portal clones the portlet for configuration [assume Consumer somehow finds the original entity]
3. getPropertyDescription(portlet clone) – the portal queries the portlet clone for the description of configurable properties
4. getProperties(portlet clone) – the portal queries the portlet clone for configurable properties' default values [steps 3&4 could be batched into a single round-trip]
5. setProperties(portlet clone, width) – the administrator sets the value of the width property [returns entity context reflecting new state]
6. setProperties(portlet clone, height) – the administrator decides that the calculated height is inappropriate and manually adjusts it (causing the width to be automatically recalculated)

7.6 Open Issues

- setProperties() needs to return an entity context
- modifyEntity() seems to be redundant with setProperties() – should drop modifyEntity() (setProperties() is more general)
- if we want to merge entity and session handles, i.e. treat a session handle as an extension of an entity handle, then we probably need to similarly extend the interaction context(s) from entityContext – this would allow setProperty to return the same structure for both entity and session scoped property operations.