Hibernation of Advanced Railroad Trains (ArrT)

Step Three – ...... and Communicate them (aCt)

SMS Facility Loaders

This hibernation report deals with the so-called Facility Loaders.

Facility Loaders are prototypes of the SMUOS framework that help to load dynamic elements, because dynamic elements are also referred to as "dynamic facilities".

Now, during the implementation of Step 0033.11, this Hibernation Report is given the 3.x version number, and Chapters 1 – 3 should already have the final content.

However, chapter 4 on "Unbound Objects (UBOs)" will still change when the final version of the software is released from Step 0033.11.

Table of Content

1 Dynamic Elements – Dynamic Facilities

1.1 Introduction

As a "facility", we actually refer to any element of an SMS - a Simple Multiuser Scene - that somehow appears to the user, be it an avatar, a module, a model, an MIDAS object, or any aggregate of information find their expression in the scene.

As "dynamic" elements, we refer to all those elements that are not automatically loaded and initialized when loading and initializing the Scene Instance, but later "loaded" in the course of the simulation.

It should be noted that there must be some **trigger** that causes the reloading of the dynamic element.

Furthermore, dynamic elements can be deleted – so to speak unloaded – long before the scene instance is destroyed. Here is the peculiarity to take into account that most X3D player, the content that is no longer needed, not immediately remove from memory, but still leave it there for a while.

That's why you have to explicitly "disable" this content before removing it from the scene so that it behaves completely "passively" and can not disturb the simulation anymore.

1.2 Requirements of the SMUOS Framework

The SMUOS Framework makes the following assumptions about all "dynamic facilities" in an SMS:

- A dynamic element can be loaded using the Browser.createVrmlFromURL () method, just the URL of the element is needed therefore
- Each dynamic element is a single X3D node that is loaded using this method (Browser.createVrmlFromURL () actually has an MFNode value as the result, but in our case only the element with index [0] is used)
- All dynamic elements of a "class of dynamic elements" can be indexed and stored in an MFNode field
- Dynamic elements can - and must - undergo "basic initialization" according to the concepts of the SMUOS framework before they can be used
- Dynamic elements can - and must - be disabled using the disable field (SFTime) before removing them from the scene

1.3 The SMS Loader

The SMS Loader is implemented in the prototype SmsLoader and provides basic support for loading dynamic elements.

The dynamic elements are stored in the field dynElems (MFNode), whereby unused indices are not assigned the value NULL, but with "any pointer".

The status of each element can be determined via the field dynElemStates (MFInt32), whereby it is guaranteed at all times that dynElems and dynElemStates have the same dimension.

The SMS Loader calls the method Browser.createVrmlFromURL (), then carries out the "basic initialization" of the element itself and then serves as a "pacemaker" for the initialization of the dynamic element, but the user essentially has to make himself, albeit clocked by the SMS Loader.

Only when the dynamic element is fully initialized can this SMS Loader load the next element. However, the SMS Loader maintains a queue in which the dynamic elements wait for loading, the loading of which has already been triggered.

When the SMS Loader gets the command to remove an item from the scene, it will make sure that the item is previously disabled

When the SMS Loader becomes disabled, it makes sure that all loaded elements are disabled and removed from the scene

1.3.1 "Basic Initialization" of the SMS Loader

Since the SMS Loader is one of many SMUOS prototypes, it also offers on its external interface the fields that all SMUOS prototypes offer, namely:

```
<!-- Common fields for the MASTER/DEP state machine →
<field accessType='outputOnly' name='sendLoaded' type='SFBool'/>
<field accessType='inputOnly' name='receivePing' type='SFBool'/>
<field accessType='outputOnly' name='sendPong' type='SFBool'/>
<field accessType='inputOnly' name='receiveBasicInit' type='SFBool'/>
<!-- Common fields for all SMUOS prototypes →
<field accessType='outputOnly' name='objType' type='SFString'/>
<field accessType='outputOnly' name='version' type='SFFloat'/>
```

The fields **objType** and **version** are there to identify one instance of the SMS Loader as such, the other four fields are for "Basic Initialization", i. the scene waits until all external prototypes are loaded before calling the **receiveBasicInit** field.

By calling the **receiveBasicInit** field, the SMS Loader tentatively sets the size of the **dynElems** and **dynElemStates** fields, based on the field

<field accessType = 'initializeOnly' name = '**typicalDynElemSpace**' type = 'SFInt32' value = '20 '/>

and outputs to the environment the actual size in the event **dynElemSpace**:

<field accessType = 'outputOnly' name = '**dynElemSpace**' type = 'SFInt32' />

Now the following two fields are initialized and can be used immediately:

<field accessType = 'inputOutput' name = '**dynElems**' type = 'MFNode'> </ field>

<field accessType = 'inputOutput' name = '**dynElemStates**' type = 'MFInt32' value = '' />

1.3.2 Initialization and disabling of the SMS Loader

The SMS Loader provides the following fields for initialization and disabling:

<field accessType = 'inputOutput' name = '**commParam**' type = 'SFNode' value = 'NULL' />

<field accessType = 'inputOutput' name = '**modParam**' type = 'SFNode' value = 'NULL' />

<field accessType = 'inputOnly' name = '**disable**' type = 'SFTime' />

Since the SMS Loader requires a reference to the Common Parameters (**commParam**), it must be initialized with either the **commParam** field or the **modParam** (Module Parameters) field before using it.

After use, you should put the SMS Loader back in a passive state with the field "**disable**". This also ensures that any dynamic elements that are still loaded are safely disabled and removed from the scene.

You can initialize the SMS Loader several times in succession and then disable it again. This can be used, for example, to "disassemble and unload" all the dynamic elements of a "class of dynamic elements" in one fell swoop.

If you initialize the SMS Loader with the **commParam**, then it stores this pointer for later use. If a disabling is currently in progress, the system first waits until disabling is completed.

If you initialize the SMS Loader with the **modParam**, then the same thing happens, but additionally the SMS Loader immediately listens to the **disable** field of the **modParam** and disables itself as soon as the module is disabled.

1.3.3 1.3.3 Loading and Initializing a Dynamic Element

To implement this function, the SMS Loader provides the following fields on its external interface:

```
<field accessType='inputOutput' name='numOfProcedures' type='SFInt32' value='0'/>
<field accessType='inputOutput' name='urls' type='MFString' value=''/>
<field accessType='inputOnly' name='loadElement' type='SFInt32'/>
<field accessType='outputOnly' name='elementQueued' type='SFInt32'/>
<field accessType='outputOnly' name='elementOccupied' type='SFInt32'/>
<field accessType='outputOnly' name='elementFailed' type='SFInt32'/>
<field accessType='outputOnly' name='initializeElement' type='SFInt32'/>
<field accessType='inputOnly' name='finishProcedure' type='SFInt32'/>
<field accessType='outputOnly' name='elementReady' type='SFInt32'/>
```

In preparation for loading and initializing a dynamic element, the user must set the following fields:

numOfProcedures ...... Number of procedures into which the initialization is to be decomposed

urls ...... this value should be passed to the method Browser.createVrmlFromURL ()

In order to actually trigger the load, the user must now set the field **loadElement** to the desired value **dynElemIdx** if he already knows the desired index of the element and to a value less than 0 if he does not yet know this index.

Then the SMS Loader inserts the information about the element to be loaded into the internal queue (**numOfProcedures**, **urls** and **dynElemIdx**) and returns in the **elementQueued** = dynElemIdx field that the element is now in the queue. The final element index is already used here, and the dimensions of the dynElems and dynElemStates fields are also increased if the index does not fit into the old dimensions.

Then, when the element has its turn, ie when the previous elements in the queue have been processed, then the element is actually loaded.

First it checks if **dynElemStates** [**dynElemIdx**] is greater than or equal to zero, ie if another element is already stored at this index. In this case, the SMS Loader aborts, reports **elementOccupied** = dynElemIdx and proceeds with the next entry in the queue.

In the good case, the SMS Loader remembers the **numOfProcedures** value for this element and actually calls the Browser.createVrmlFromURL () method.

Now, when the element is actually loaded, it is stored in the **dynElems** field and undergoes basic initialization. To do this, each dynamic element must support the four fields **sendLoaded**, **receivePing**, **sendPong**, and **receiveBasicInit**.

Now that the element is actually loaded and after the "Basic Initialization" has been successfully completed, the element is initialized.

Initialization is done in **numOfProcedures** steps, counting down in **dynElemStates** [**dynElemIdx**].

If the value 0 is in **dynElemStates** [**dynElemIdx**], then the dynamic element has been successfully loaded AND initialized.

For each step (for each "procedure") performed during initialization, the following happens (if **numOfProcedures** was 0 then it never happens):

1. With the field **initializeElement** = dynElemIdx the SMS Loader requests the user to execute a procedure (ie a step of initialization)

2. The procedure is identified by the value **dynElemStates** [**dynElemIdx**], counting down to one for the steps (procedures) of **numOfProcedures**.

3. The dynamic element is already in its place in the field **dynElems** [**dynElemIdx**].

4. After completing the procedure (step), the user must use **finishProcedure** = dynElemIdx to tell the SMS Loader that the procedure is finished and that he should continue with the next procedure

After the user has performed all the procedures of the initialization, the SMS Loader reports with **elementReady** = dynElemIdx that the element can now be used.

In the event that the user wants to add the dynamic elements to the scene, for example as children of a <Group> node, because it is not enough to have the dynamic elements in the **dynElems** field, the SMS Loader offers some fields which can be routed directly to the fields of a <Group> node.

These are described in chapter 1.3.5.

1.3.4 Disabling and unloading a dynamic element

For disabling and unloading dynamic elements, the SMS Loader offers the following field:

<field accessType = 'inputOnly' name = '**discardElement**' type = 'SFInt32' />

Through an event **discardElement** = dynElemIdx, the user tells the SMS Loader to disable and unload a dynamic element.

1.3.5 Interaction with a <Group> node within the scene

The SMS Loader provides the following elements for interacting with a <Group> node within the scene:

```
<field accessType = 'outputOnly' name = 'initializingElement' type = 'MFNode' />
<field accessType = 'outputOnly' name = 'addElement' type = 'MFNode' />
<field accessType = 'outputOnly' name = 'removeElement' type = 'MFNode' />
```

The **addElement** and **removeElement** fields can be directly linked to the **addChildren** and **removeChildren** fields of a <Group> node.

The **addElement** field has the property that dynamic elements are added to the scene only after they have been initialized.

If you use the **initializingElement** field instead of the **addElement** field, then the dynamic elements are already added to the scene before they have been initialized.

2 Module Related SSC Dispatcher

The SSC Base itself is a user of the SMS Loader, as can be easily recognized by the following excerpts from the file SscBase.x3d.

```
<ProtoDeclare name = 'SscBase'>
  <Proto Interface>
    ... …
  </ Proto Interface>
  <Proto Body>
    ... …
      <Script DEF = 'SimpleSceneControllerBase' directOutput = 'true' mustEvaluate = 'true'>
        ... …
        <field accessType = 'inputOutput' name = 'dispatcherLoader' type = 'SFNode'>
          <ProtoInstance DEF = 'DispatcherLoader' name = 'SmsLoader'>
            <fieldValue name = 'numOfProcedures' value = '1' />
            <fieldValue name = 'urls' value = '"../ sms / SscDispatcher.x3d"' />
          </ ProtoInstance>
        </field>
```

Namely, he uses the SMS Loader to dynamically load a so-called "Module Related SSC Dispatcher" for every registered module and to save it in the **commParam.sscDispatchersGroup** field.

If the module is deregistered, then also the SSC dispatcher is disabled and unloaded.

3 Dynamic Modules

The SMUOS framework provides the prototype SmsModuleLoader, so the framework has some help in loading and unloading dynamic modules.

The SMS Module Loader provides the following fields on its external interface:

<field accessType = 'inputOutput' name = '**dynModConf**' type = 'SFNode' value = 'NULL' />

<field accessType = 'inputOutput' name = '**commParam**' type = 'SFNode' value = 'NULL' />

<field accessType = 'outputOnly' name = '**registerModules**' type = 'MFString' />

<field accessType = 'inputOutput' name = '**registeredModules**' type = 'MFString' value = '' />

<field accessType = 'inputOnly' name = '**loadModuleWrappers**' type = 'MFString' />

<field accessType = 'outputOnly' name = '**moduleWrapperLoaded**' type = 'SFNode' />

<field accessType = 'inputOnly' name = '**loadModule**' type = 'SFNode' />

<field accessType = 'inputOnly' name = '**unloadModuleWrapper**' type = 'SFInt32' />

The field **dynModConf** ("Dynamic Module Configuration") must be set by the user, it contains a node whose fields contain the information about all dynamic modules.

The fields **commParam**, **registerModules** and **registeredModules** are best connected to the fields of the same name of the SSC Base:

Once the SSC is initialized, it reports the Common Parameters (**commParam**). With the **commParam** now also the SMS Module Loader is initialized, reads the contents of the "Dynamic Module Configuration" and registers the dynamic modules with **registerModules**.

Throughout the simulation, the SSC keeps the list of registered modules currently in the **registeredModules** field, which contains both dynamic and static modules.

3.1 Loading a dynamic module

If the frame decides to load a registered dynamic module, then it must pass the module name to the SMS Module Loader in the **loadModuleWrappers** field.

This will ensure that the module, if it was already loaded, initially unloaded and then freshly loaded.

As soon as the module wrapper has been loaded, the SMS Module Loader logs in with the field **moduleWrapperLoaded**. The value of this event points to the module wrapper and allows the frame to set the initial state of the module wrapper. Once that's done, the frame must "mirror" the value back to the **loadModule** field.

Now the module is actually loaded.

3.2 Disabling and unloading a dynamic module

The index of a module (the so-called **moduleIx**) is the index at which the SSC outputs the module name in the field **registeredModules**.

Now, if the frame wants to unload a dynamic module, it must pass the module's **moduleIx** to the **unloadModuleWrapper** field.

Furthermore, the SMS Module Loader automatically unloads a dynamic module when it is loaded and when the SSC deletes the module name from the **registeredModules** field (if it is deregistered).

4 Unbound Objects (UBOs)

THIS CHAPTER CONTAINS UNSETTLED FUTURE CONCEPTS. IT WILL BE TRANSLATED AND UPDATED WITH RELEASE „ARIMATHEA“ (0033.11)

UBOs beziehen sich immer auf eine Universal Object Class (UOC). Eine UOC ist so etwas wie eine „grobe Klassifizierung von Objekten“, aus der man darauf schließen kann, welche SSC Extension man in seiner Szene benötigt, um eben eine bestimmte UOC zu unterstützen.

Die grundlegenden Ideen über UOCs und Object Types (Ots) **finden sich in Kapitel 4.1** .

Die UOCs werden von SSC Extensions definiert. Das heisst also, dass jeder Programmierer, der eine oder mehrere neue UOCs anbieten möchte, zuerst einmal eine SSC Extension programmieren muss.

UBOs und UOCs werden aber vom SMUOS Framework unterstützt. Das heisst, der Programmierer, der eine neue UOC für UBOs anbieten möchte, kann auf folgende X3D Prototypen zurückgreifen:

- UOC Dispatcher (SscDispatcher.x3d)
- UBO Loader (SscUboLoader.x3d)
- UBO Loader Stub (SmsUboLoaderStub.x3d)
- Basic MIDAS Objekt „Creator“ (MoosCreator.x3d)

Wie diese drei Prototypen zusammenarbeiten, und was noch fehlt, um eine UOC für UBOs zu definieren, **ist im Wesentlichen in Kapitel 4.2 beschrieben**.

Beide Unterkapitel sind in englischer Sprache gehalten, da sie aus einem anderen Paper stammen, welches auf einem englischen Blog veröffentlicht worden war.

4.1 Universal Object Classes (UOCs) and Object Types (OTs)

If you want to create a UBO, then it's not enough to know the UOC. A UOC is only a "rough classification" of an object, e.g

- military helicopter
- rail vehicle
- sports car
- propeller airplane
- and so on

An Object Type (OT) provides the possibility to categorize an object and it provides the possibility to load an object from a URL

- "Boeing AH-64 Apache", cat: "attack,1-rotor", URL: "http://x3d.net/apache.x3d"
- "ÖBB Rh 1044", cat: "1435mm,electric,BoBo", URL: "http://x3d.net/loco1044.x3d"
- "Bugatti-57G", cat: "4743cm3,Inj8,160PS", URL: "http://x3d.net/bugatti57g.x3d"
- "Cessna-182-Skylane-RG", cat: "4seat,1motor", URL: "http://x3d.net/cessna.x3d"

An Object Type is identified within an SMS by "UOC Name + Object Type ID"

- The UOC Name is an SMS wide unique identifier of a UOC
- Object Type IDs are unique within each UOC within an SMS
- Object Type IDs are globally registered in the global state of the UBO Loader (Object Type List – OTL) and they are identical in all scene instances of a multiuser session
- When an OT is deregistered, then all UBOs that had been created from this OT, are deleted
- Categories and URLs are defined in the DED (Dynamic Element Definition), which might be different in each scene instance. The categories and URLs are stored locally in the UBO Loader of each scene instance.
- If two scene instances use different URLs or different categories for their UBOs, then we say the scene instances provide different "views" to their users.

4.2 Architecture for Unbound Objects (UBOs)

Pre-Conditions and Basic Assumptions

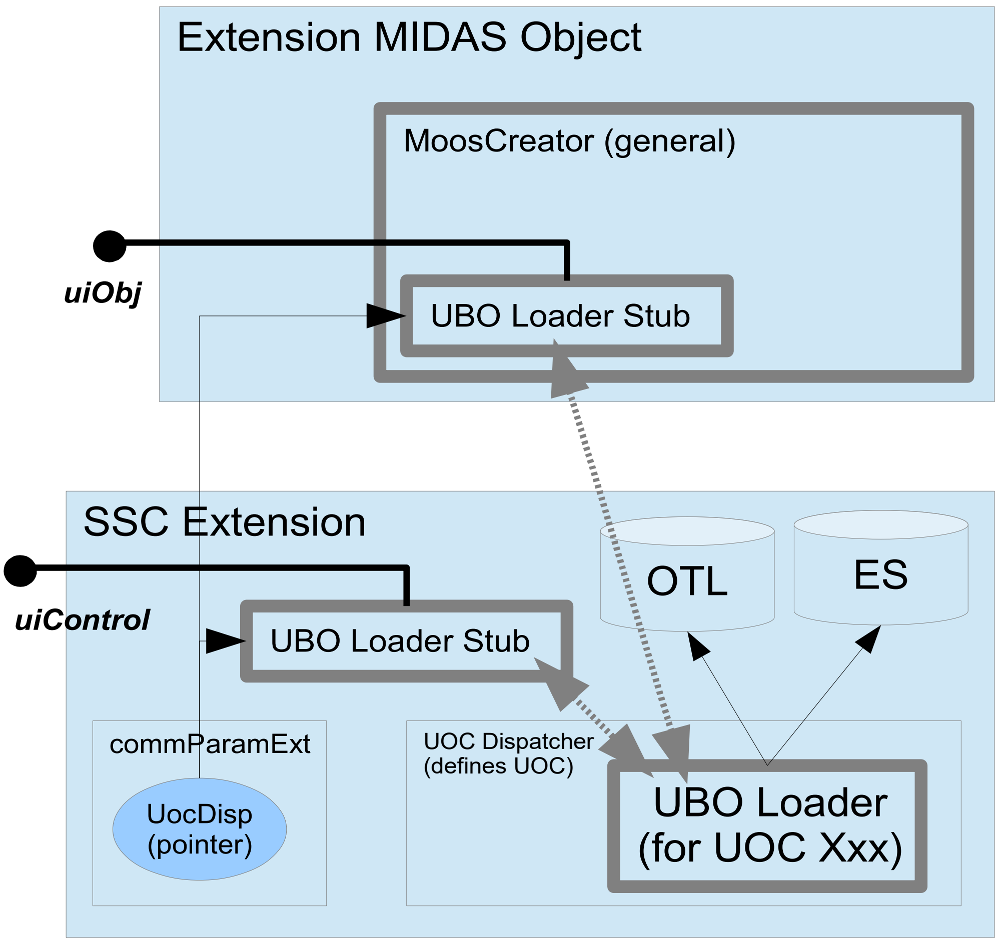
- The MIB supports the **Modes Of Operation** (MOOs) I, II, III, IV and V, where the MOOs III, IV and V apply to unbound objects
    - MOO "LOADED"..........the object has been freshly loaded
    - MOO III "initialized"…..the object has been initialized or it has been deAttached
    - MOO IV "attached"........the object has been attached to a module
    - MOO V "disabled".........the object has been deAttached and deInitialized (if it was "attached") or it has been deInitialized (if it was "initialized") or it came directly from MOO "LOADED"
- The SSC Core and the SSC Base support the extension of the SSC by **SSC Extensions**
- Each SSC Extension can define **Universal Object Classes** (UOCs) by instantiating UOC Related SSC Dispatchers
    - UOCs are rough classifications of objects, e.g. "RailVehicles", "MilitaryHelicopters" or "Rockets"
- If an SSC Extension wants to make a UOC support UBOs, then it **must** instantiate one **UBO Loader** as a part of the UOC Related SSC Dispatcher
- The **UBO Loader** cooperates closely with **MoosCreator MIDAS Objects**, which are some means to be able to handle UBOs via the uiObj interface of Creator objects.
- Additionally the SSC Extension **may** instantiate one **UBO Loader Stub** to be able to handle UBOs by means of the uiControl interface.

Object Types (OTs) and Object Type Identifiers (OTIs)

- Each UBO is created based on an **Object Type** (OT)
- Where Universal Object Classes (UOCs) are rough classifications of objects (e.g. "rail vehicle", "helicopter" or "rocket"), an Object Type is a clear definition of a type of an object
    - OTs are e.g. "OeBB1044", "Apache" or "SaturnV", sticking to the above example
- The **Basic MIDAS Object MoosCreator** will support the lifecycle of all UBOs. Each instance of the MoosCreator supports one and only one UOC, it provides the current list of Object Types and Categories (see below) and enables the user to influence the lifecycle of UBOs
- The Object Types, the Categories and the Object IDs of the UBOs will be unique within each UOC, e.g.
  <objTypeId> = "Uoc.Base.RailVehicles.OeBB1044",
  <category> = "Uoc.Base.RailVehicles.1435mm",
  <extObjId> = "Uoc.Base.RailVehicles-1044_212"

Basic Elements of the Architecture for UBOs

- The UBO Loader
- The UBO Loader Stub
- The MoosCreator



Registration of the MoosCreator at the UBO Loader (establish association)

1) The SSC Extension defines a UOC by a UOC Related SSC Dispatcher ("UOC Dispatcher").
2) The SSC Extension makes the UOC support UBOs by a UBO Loader. The UBO Loader maintains two global states – the Object Type List (OTL) and the Existence State (ES).
3) The lifecycle of the UBOs can be controlled via the uiObj interface of the MoosCreator.
4) However, the MoosCreator cannot access the UBO Loader per se. First the SSC Extension must provide and Extension MIDAS Object that can access the pointer to the UOC Dispatcher. With this pointer, the MoosCreator can access the UBO Loader.
5) The MoosCreator uses an UBO Loader Stub to connect to the UBO Loader.
6) The SSC Extension can use a UBO Loader Stub to make it possible to control the lifecycle of UBOs via the uiControl interface of the SSC Extension.

Duties of the UBO Loader, of the UBO Loader Stub and of the MoosCreator

- The UBO Loader maintains a list of all **Object Types** of one UOC (Object Type List OTL)
    - At the Layout Description File (LDF), the Object Type is addressed by **UOC Name** AND **Object Type ID**, e.g. "Uoc.Base.RailVehicles.OeBB1044"
    - At the UBO Loader Stub, the Object Type is addressed by the **Object Type ID** only, e.g. "OeBB1044"
    - An object type is locally associated with zero or more **Categories**
    - Categories are used to filter lists of Object Types
    - The UBO Loader keeps all **connected UBO Loader Stubs** up to date about the list of Object Types. Therefore it reads the filter from the UBO Loader Stub, filters the current list of Object Types and outputs the filtered list to the UBO Loader Stub.
- The UBO Loader together with the MoosCreator/UBO Loader Stub care for the lifecycle of UBOs
- Each UBO Loader maintains the global **Existence State (ES)** of all UBOs of one UOC
    - List of all **<objId>**s at index **objectIx**
    - List of all **<objectTypeIx>**s at index **objectIx**
    - List of all **<objectState>**s at index **objectIx**
- The **objectIx** is a globally valid index of each UBO within the Existence State
- Each UBO can be identified within an SMS by UOC Name + <objId> or by UOC + objectIx
- The objectIx can be used at the interface of the UBO Loader Stub to identify one UBO of the given UOC
- The UBO Loader keeps all **connected UBO Loader Stubs** up to date about the list of UBOs

When CREATING a UBO (this is a global procedure)

- The MoosCreator or the SSC Extension ensures to have all data for "ObjectTypeIx + Initial State"
- The "creating instance" of the UBO Loader Stub "occupies" the UBO Loader globally; "occupying" is a global process, that means: only one instance of the UBO Loader Stub within the whole UOC – and within all scene instances – can occupy the UBO Loader at a time
- The UBO Loader Stub may suggest an <objId> for the creation of the UBO and then it triggers the creation
- The <objId> and the <objectTypeIx> are set to their correct values globally in the ES and the <objectState> is set to CREATING globally;
  only one objectIx can be CREATING at a time (in each UOC)
- The creating instance of the UBO Loader recognizes "his" sessionId is CREATING now and hence it loads the FIRST instance of the requested UBO Wrapper
- When the FIRST instance of the UBO Wrapper has been loaded, then the UBO Loader invites the CREATING instance of the UBO Loader Stub to brute force initialize the global Wrapper State to "initialState" + "moduleIx" + "firstInstanceFlag=true"
- Now this UBO Wrapper can be accessed by the SSC Extension with the help of the objectIx
- If the creation was triggered by a MoosCreator, then the MoosCreator announces all his instances for progress reports (this is possible, because he knows the objectIx now)
- Now the UBO Loader sets the <objectState> to CREATED and it finishes the occupation
- All other instances of the UBO Loader recognize the state CREATED and hence they load the UBO Wrappers, too
- Now all instances of the UBO Wrapper can be accessed by the instances of the SSC Extension with the help of the objectIx
- Those scene instances, where "delayed loading" happens, set progress to "100%".

When LOADING a UBO (this is a local procedure)

- If a UBO Wrapper has been loaded and if the <objectState> at the objectIx is CREATED, then the SSC Extension may decide to actually load the local instance of the UBO
- If the decision is for "delayed loading", then the progress is set to 100%, otherwise the UBO Wrapper cares for the display of the progress, if the MoosCreator has announced himself for progress reports
- The first instance of the UBO that is actually loaded (in whatever scene instance) recognizes that it is the first instance by the <firstInstance=true> flag in the UBO Wrapper and hence brute force initializes the state of the UBO by "Initial State". It deletes the <firstInstance=true> flag in the UBO Wrapper
- After the UBO reports the state has been "brute force" or "just" initialized (new output field?), then the UBO Wrapper sets the "SELF" field at the uiObj interface, evaluates the "moduleIx", searches for the modParam (if moduleIx >= 0), and attaches the UBO to the module (if the moduleIx >= 0 and if the module is loaded and initialized).
- In case of UBOs the MIB Core prototype adds the SELF node to the <Group> node in the module coordinator

When a module has been loaded and initialized (this is a local procedure)

- The ModCoord Extension searches all relevant UBO Loaders for UBO Wrappers with the own moduleIx
- If wrappers are found and if the contained UBOs are loaded, then the UBOs are attached to the module
- In case of UBOs the MIB Core prototype adds the SELF node to the <Group> node in the module coordinator

When a module has been disabled

- The ModCoord Base sends commParam to all UBOs in the <Group> node. The MIB Core prototype removes the SELF node from the <Group> node

Module Activation/Deactivation

- All UBOs that are attached to the module, receive the module activity and react accordingly.

When UNLOADING a UBO (this is a local procedure)

- If the <objectState> of a UBO is CREATED and if the UBO has been loaded locally, then the SSC Extension can decide to unload the UBO locally (without influencing the Existence State)
- The UBO Wrapper is told that it should unload the UBO and it is done
- The MIB Core prototype removes the SELF node from the <Group> node in the MC.
- Loading the same UBO again locally is possible, as long as the <objectState> is CREATED

When DELETING a UBO (this is a global procedure)

- Deleting a UBO is a service of the UBO Loader. Hence it can be called
  - by the SSC Extension
  - by any UBO of the same UOC (via „universalObjectClass“)
  - by the MoosCreator (MoosCreator has a list of all UBOs)
  - Delete „Self“ is a Service of MIB Core
  - By the Console Interface
- The UBO Loader may decide to delete a UBO, when the <objectState> is CREATING or CREATED
- The UBO Loader will set the <objectState> to DELETED and keep the <objId> and the objectIx reserved for 20 seconds
- All Instances of the UBO Loader detect that the state changed to DELETED and hence the UBO Wrappers are disabled and unloaded – so that they unload the UBOs (see above)

When Deregistering a Module (this is a global procedure)

- The central server of SSC Base detects, when he deregisters a module in the commState. Then he informs all UBO Loaders to delete all UBOs of that module (set moduleIx = -1)

4.3 Fields of UboLoaderStub

- **„universalObjectClass“ (SFNode)**
  This field takes the pointer from the SSC Extension (commParamExt) that points to the UOC Dispatcher.
  This enables to the UBO Loader Stub to connect to the UBO Loader.
  The same value will be written to the uiObj Interface of the UBOs, after they will have been loaded by the UBO Wrapper
- **„disable“ (SFTime)**
  An event „Now“ at this field will disconnect the UBO Loader Stub from the UBO Loader and it will disable the UBO Loader Stub. A new initialization with „universalObjectClass“ will be possible
- **„categories“ (MFString)**
  This field will be used as filter for the display of Object Type IDs
- **„objectTypeIds“ (MFString), objectTypeIxs (MFInt32)**
  These fields are the filtered Object Types that can be created by this UBO Loader Stub.
  They are the „filtered OTL“ of the connected UBO Loader
- **„objects“ (MFNode), objectsStates (MFString)**
  This is a „mirror of the ES“ of the connected UBO Loader
  „objects[ix]“ contains the <objId> and the <objectTypeIx> (actually they are the UBO Wrappers)
- **„preferredObjId“ (SFString)**
  Before starting the creation of a UBO, here a specific „objId“ can be requested
- **„createObjectType“ (SFString), „createObjectTypeIx“ (SFInt32)**
  start the creation of an UBO
- **„initializeObjectIx“ (SFInt32)**
  This UBO Wrapper shall be brute force initialized (< 0 --> Error)and then objectIxInitialized shall be triggered
- **„objectIxInitialized“ (SFInt32)**
  Brute Force Initialization done --> free the UBO Loader again