MIDAS Best Current Practices

This hobby report is a "snapshot" and will not be updated any further. It was written at the end of step 0033.10 "Pieta".

Version 4.3 is indicating the version at the first serious specification of the release "Arimathea" (step 0033.11), where the basic concepts of "Arimathea" seem to be stable now. No update done.

1 Overview "How to make a MIDAS object"

To make your own MIDAS objects, you have to stick to some rules.

On the one hand, MIDAS objects can use the lower layers of the SMS framework, i.e.

- the module coordinator MC (via the external interface eiMod),
- the Simple Scene Controller SSC (via the external interface eiControl) and
- the SSC Dispatcher DISP (via the eiDisp external interface).

On the other hand, the SMS Framework requires that MIDAS objects offer certain services,

- that is the definition of the user interface uiObj.

For all these tasks MidasBase helps the programmer of the MIDAS object (MOB).

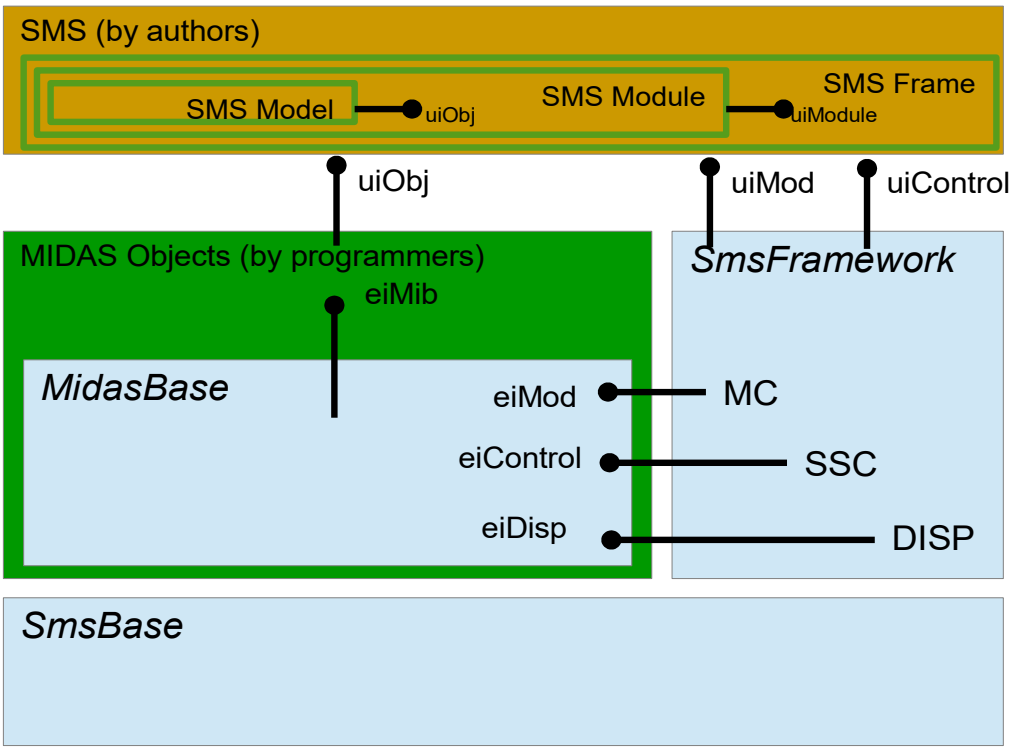

Figure 1: MidasBase as a link between SmsFramework and MIDAS object

2 2 What the heck are MIDAS objects?

Okay, there is the historical explanation:

- In the beginning I decided to set up the SIMUL-RR project (then it was not called SrrTrains v0.01) on the network sensor
- I had to tinker around the network sensors of two manufacturers own prototypes, as these had ever been defined differently by the two companies, but I wanted to tinker scenes that should run on more than one Web3D browser (what else would the whole overhead with an ISO standard, if I may ask)
- These self-made prototypes got more and more features over time, so they became their own concept
- They became the so-called SRR objects (Simulated Railroad Objects)
- Then at some point I made my own project out of the base module of the SRR framework (that was a collection of prototypes that should support the SRR objects) and made some renames:
    - the "SRR Controller" became the "Simple Scene Controller"
    - the "SRR Module Coordinator" became the "SMS Module Coordinator"
    - the Base Module of the SRR Framework became the SMUOS Framework
    - The SRR objects were also renamed to MIDAS objects

OK, but if I buy such a MIDAS object, what can I do with it?

- Well, suppose you are a Web3D author and you want to build the model of a car
- Suppose further, this model should be executable in a multi-user scene by another author
- THEN you'll be looking for ways to outsource the entire multi-user problematic. You will be looking for ways to make a model of a car, regardless of whether the scene is multi-user capable or not
- THEN you'll be looking for ready-made mechanisms to simulate the car's drive, steering, and brake, and you'll want to spare yourself all the math
- AND THAT'S exactly what the MIDAS objects offer
    - MIDAS objects are highly specialized powerhouses that have all the maths of multiuser simulation under control
    - MIDAS objects can not be rendered, they remain invisible, inaudible, imperceptible, the outward appearance, the "look and feel" of your models is still completely in your hands
    - MIDAS objects are the "invisible engines" of your multiuser-capable, interactive and animated models

3 Behavior of MIDAS objects

3.1 Initialization with the Common Parameters

Because of the "MMF paradigm" we want to stick to, the scene is made up of individual modules (this can be thought of as landscape tiles) and these modules are "inhabited" by models.

The models in turn contain the MIDAS objects (MOBs).

But there is also - admittedly a bit abstract – some "all around", a "something" needed to hold the modules together and "hang them up" in a scene.

This "something" we call the "frame".

Now we also know why the paradigm is called "MMF paradigm" because it's about models in modules in a frame.

Because the frame is basic, it also includes a basic part of the SMS Framework, namely the Simple Scene Controller (SSC).

The Simple Scene Controller can access all parts of the SMS framework as well as the MIDAS objects.

So that all parts of the scene can access the SSC, it publishes a pointer to the Common Parameters (commParam) after its initialization.

To initialize an MIDAS object, you pass this pointer to it.

With the help of this pointer

- each MIDAS object can access the fields of the SSC,
- Each MIDAS object can log on to the events that the SSC broadcasts via the commParam.
- Some MIDAS objects require the SSC to access directly and individually. That means he needs pointers to these objects. This is described in chapter 3.4 "Announcement with the Simple Scene Controller".

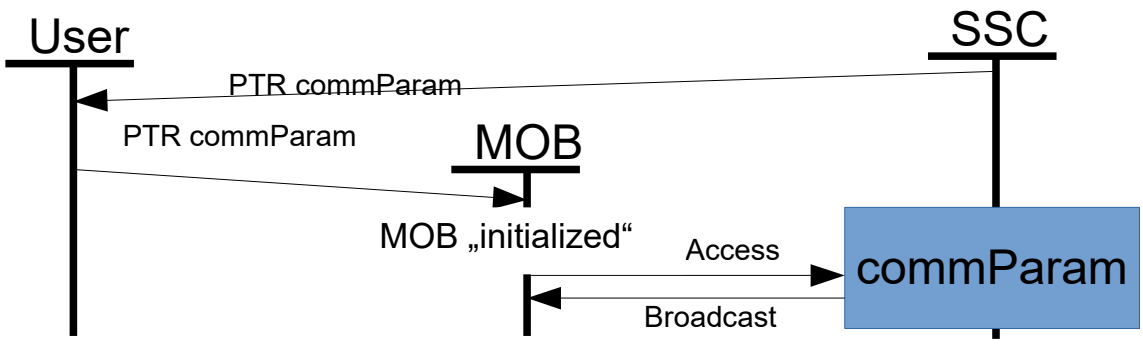

Figure 2: Access to the SSC by an initialized MOB

3.1.1 Error during initialization

As we will see in chapter 3.3 "The fields" initialized "and" attached "as well as" getScript "", after initialization a MIDAS object gives a feed back with the field "initialized".

If an error occurs during initialization, the MOB in this field sends the value NULL. Nevertheless, the initialization is considered completed (the MOB is in the status "fail-initialized") and is undone by a de-initialization.

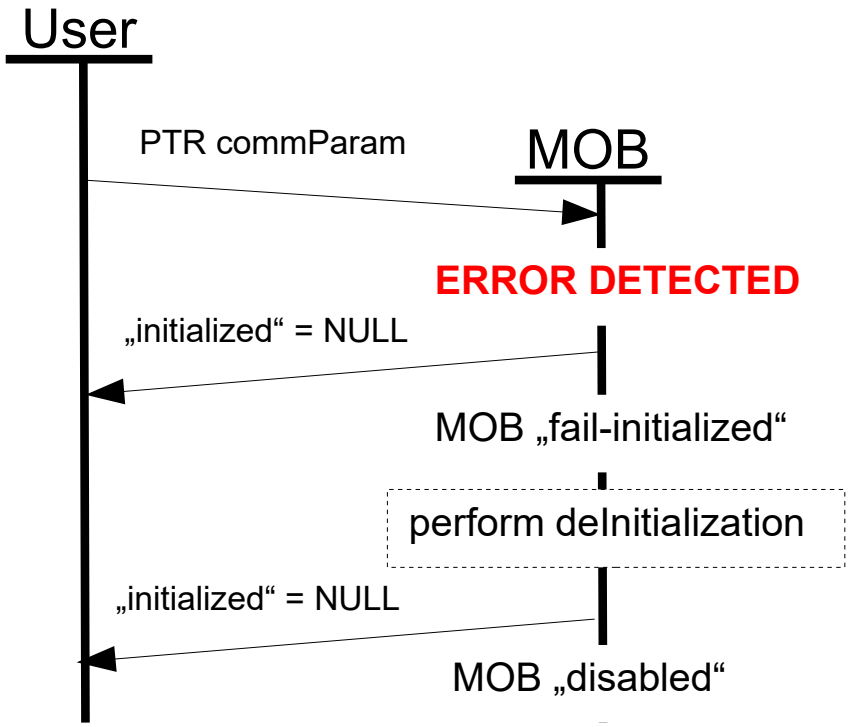

Figure 3: Initialization: error during initialization

3.2 Attachment with the module parameters

Each module contains an instance of the module coordinator (MC). This is initialized after the SSC by giving it a pointer to the Common Parameters (commParam) and then announcing itself to the SSC (so that the SSC can access all MCs individually and directly).

After the module has reached the "attached" state in this way, the MC publishes a pointer to the module parameters (modParam).

This pointer is passed on to a MIDAS object (MOB), so that it is attached to the MC.

With the help of this pointer

- Each MIDAS object can access the fields of its MC
- For example, each MIDAS object can log on to the events that the MC broadcasts via the modParam
- For some MIDAS objects, the MC must access individually and directly. That means he needs pointers to these objects. This is described in chapter 3.5 "Announcement at the Module Coordinator".

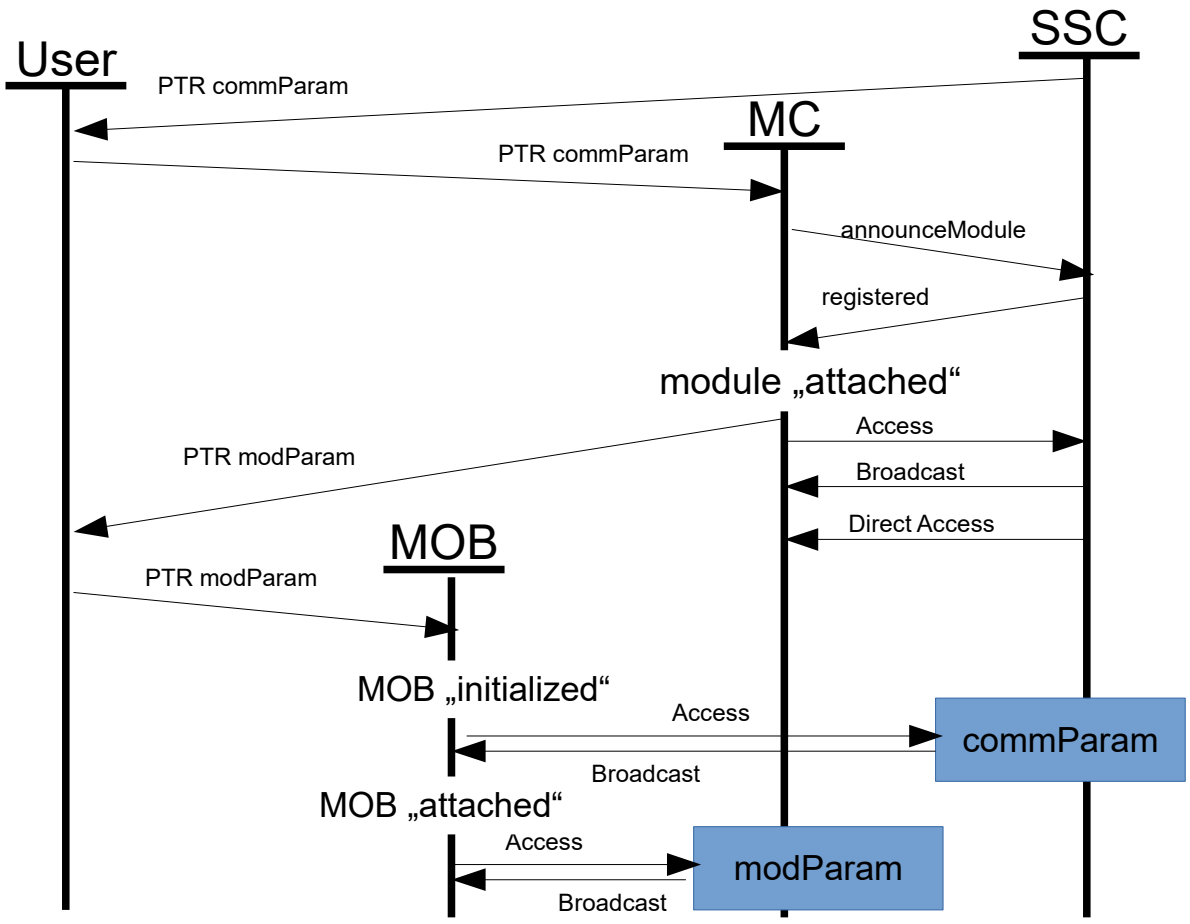

Figure 4: Access to the MC by an attached MOB

3.2.1 Error during initialization

As we will see in chapter 3.3 "The fields" initialized "and" attached "as well as" getScript "", after initialization a MIDAS object gives a feed back with the field "initialized" and after its attachment a feed back with the field " attached".

If an error occurs during initialization as part of an attachment, the MOB sends the value NULL in the "initialized" field. Nevertheless, the attachment is also continued (the MOB is then in the status "fail-attached") and then undone by a de-attachment and a de-initialization.

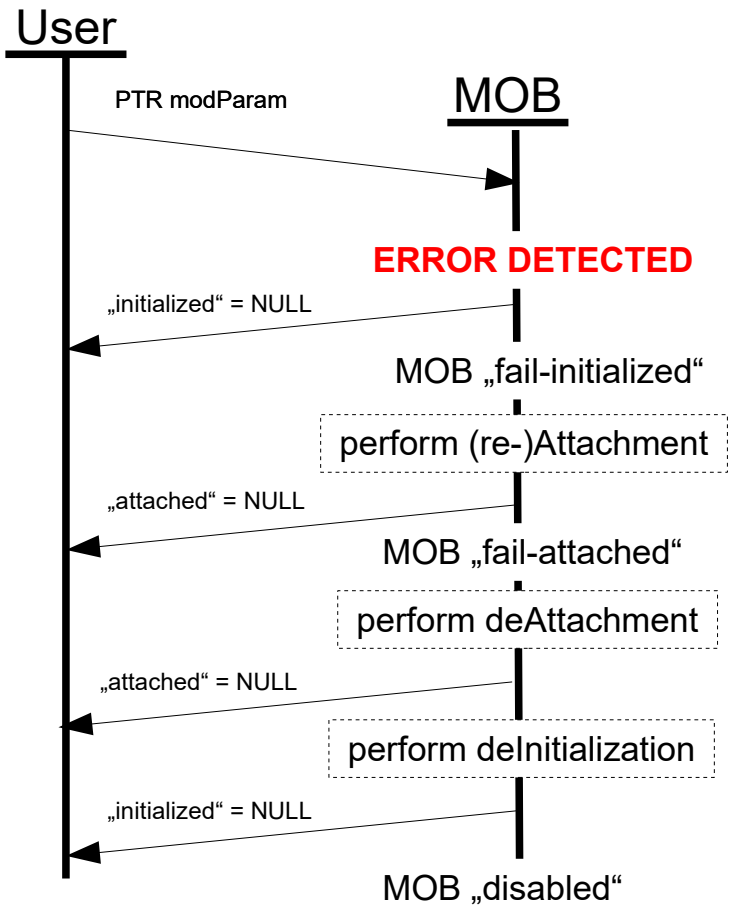

Figure 5: Attachment: error during initialization

3.2.2 Error during the attachment

As we will see in chapter 3.3 "The fields" initialized "and" attached "as well as" getScript "", after initialization a MIDAS object gives a feed back with the field "initialized" and after its attachment a feed back with the field " attached".

If an error occurs during the attachment, the MOB sends the value "NULL" in the "attached" field. Nevertheless, the attachment is continued (the MOB is then in the status "fail-attached") and then undone by a DeAttachment and a de-initialization.

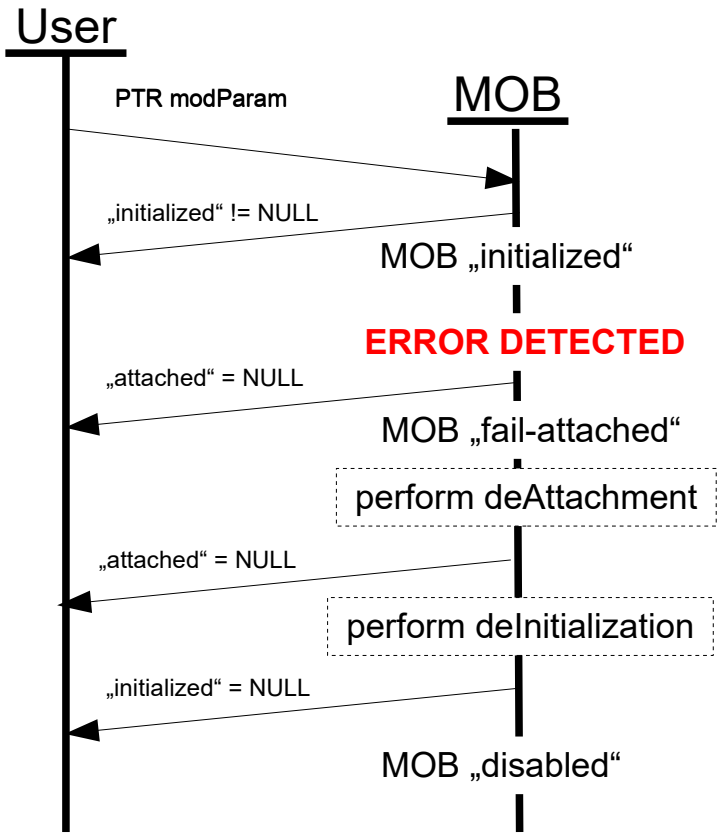

Figure 6: Attachment: error during attachment

3.3 The fields "initialized" and "attached" as well as "getScript"

3.3.1 "getScript"

Each MIDAS object contains (at least) one <Script> node, in which the functionality of the object is procedurally described.

Some of these procedures need to bring the MIDAS object to another node of the scene.

It would be nice if there was a command within a <Script> node that points to the surrounding <ProtoInstance> in a similar way to SELF in other programming languages.

Such a mechanism I did not find in X3D, but at least there is the possibility to use a pointer to the <Script> node itself in the sense of SELF.

So we define: Within the SMS framework, an MIDAS object is represented by an SFNode value pointing to the <Script> node contained in the MIDAS object. This also applies to other <ProtoInstance> s used in the context of the SMS Framework.

The best way to see this is an example: the module coordinator contains the following <Script> nodes (shown in excerpts):

```
        <Script DEF='SmsModCoordScript' directOutput='true' mustEvaluate='true'>
          <field accessType='inputOutput' name='commParam' type='SFNode' value="NULL"/>
          <field accessType="inputOutput" name="getScript" type="SFNode">
            <Script USE="SmsModCoordScript"/>
          </field>
.....
<![CDATA[
ecmascript:
.....
// Announcement protection timer expired (fires isActive = false)
function timerExpired(Value,timestamp)
{
.....
    commParam.sscBase.announceModule = getScript;
.....
}
]]>
        </Script>
```

Here you can see that the <Script> node stores the values "commParam" and "getScript" in its fields. Where "commParam" points to the common parameters and "getScript" is a pointer to the

<script> node itself.

The code "commParam.sscBase.announceModule = getScript" thus sends a pointer to the <Script> node of the module coordinator to the field "announceModule" of the <Script> node pointed to by the field commParam.sscBase.

3.3.2 "initialized" and "attached"

So that now also the user of the MIDAS object can store the pointer on the MIDAS object, in order to use it then later for example against the SSC as identification of the MIDAS object, is defined:

1. A MIDAS object that has undergone a successful (re) initialization must output a pointer to the internal <Script> node on the "initialized" field, which is also used in relation to the MC, the SSC and the DISP.
2. A MIDAS object that has undergone a de-initialization or an unsuccessful (re) initialization has to output the value NULL at the field "initialized".
3. A MIDAS object that has undergone a successful (re-) attachment must output a pointer to the internal <Script> node on the "attached" field, which is also used in relation to the MC, the SSC and the DISP.
4. A MIDAS object that has undergone a deAttachment or an unsuccessful (re-) attachment has to output the value NULL at the field "attached".
5. This "SELF" value is also used when a MIDAS object "announces" on the MC, SSC or DISP.

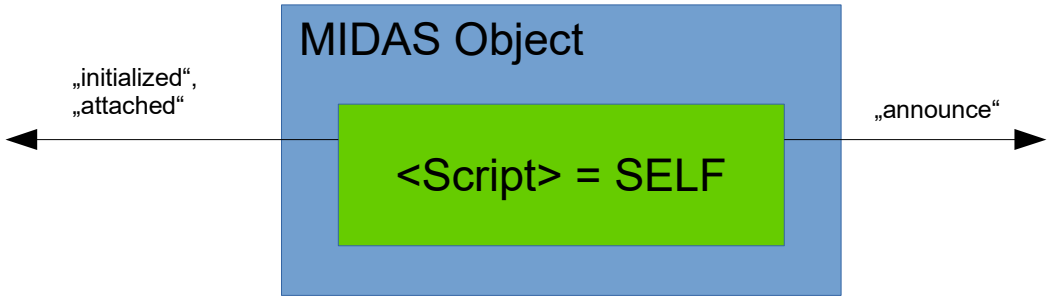

Figure 7: Usage of the <Script>-Node for SELF

3.4 Announcement at the Simple Scene Controller

An MIDAS object announcing itself to the SSC should announce this during its (re-) initialization.

This ensures that the SSC already knows the object when it outputs its "SELF" pointer on the "initialized" field.

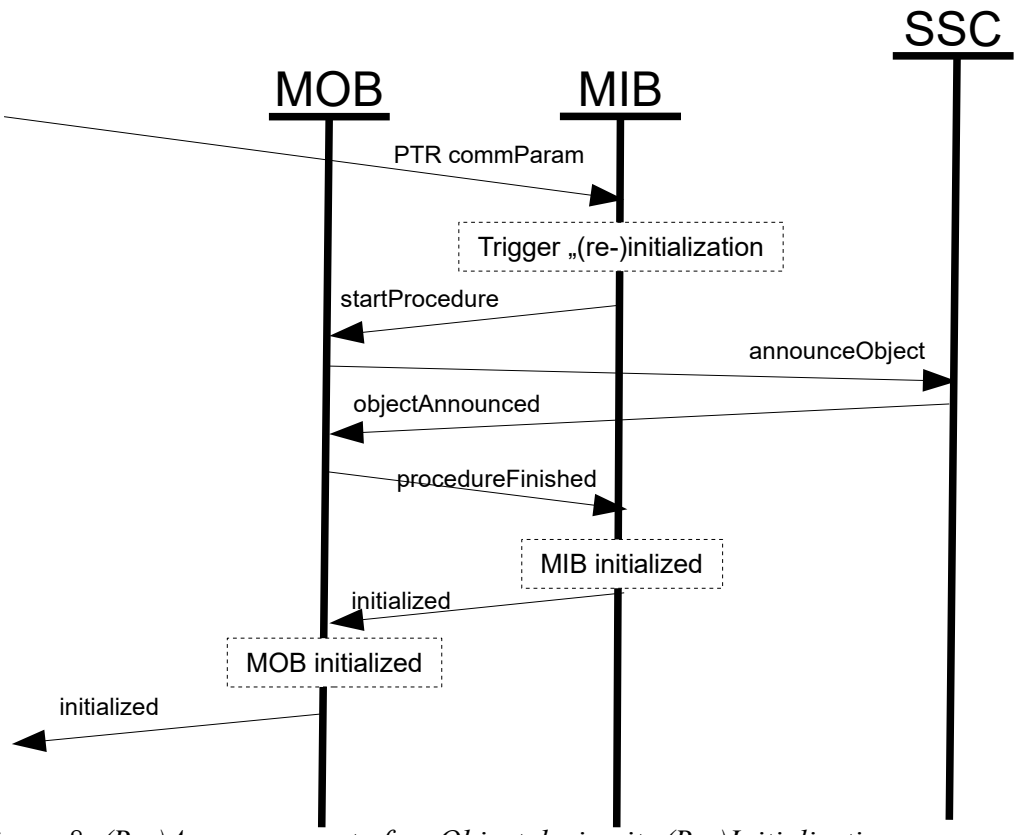

Figure 8: (Re-)Announcement of an Object during its (Re-)Initialization

3.5 Announcement to the Module Coordinator

A MIDAS object announcing itself to the MC should announce this during its (re-) attachment.

This ensures that the MC already knows the object when it outputs its "SELF" pointer on the "attached" field.

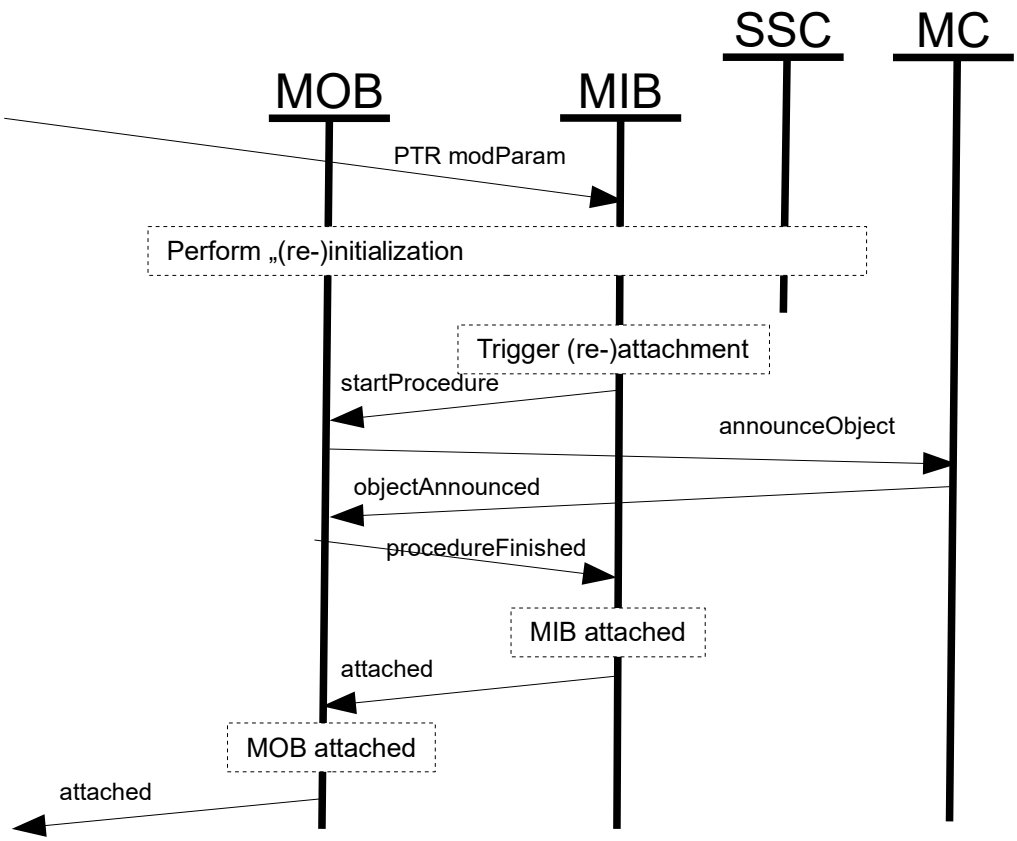

Figure 9: (Re-)Announcement of an Object during its (Re-)Attachment

3.6 Nested MIDAS objects

Basically, the MIDAS Base (MIB) ensures that all MIDAS objects (MOBs) of a model always have the same Mode of Operation (MOO).

In particular, this is a task of the MibCore prototype:

- Dependent MOBs referenced at the time of "basic initialization" in the "dependentMobsIn" field are added with addMob (). That is, they are included in the "internal list of dependent MOBs" and the first initialization or when switching to MOO V ensures that they have the same MOO as the parent object.
- AddDependentMobsIn () can also add one or more MOBs as dependent MOBs. This is always possible if no procedure is currently running. After adding the MIDAS object, you must start a MOO change on the parent object so that the child object receives the same MOO. This happens at the beginning of the (re) initialization of the parent object or after the de-initialization of the parent object.
- By removing one or more dependent MOBs with removeDependentMobsIn () - whichever is possible when no procedure is currently running - these MOBs are removed from the "internal list of dependent MOBs" and then automatically "disable" them MOO V spiked. Any own pointers to the MOB must be removed by the user himself.
- If a child object is in the "internal list of dependent MOBs" and has been disallowed for some reason ("enabledOut" fires the value "false"), then the parent object is also disablet. In the case of the parent object, this is considered a so-called "internal trigger for disabling".
- An "internal trigger for disabling" means that any existing own parent object is immediately (!) Notified, too disabling, without waiting for the trigger "enabledOut" = "false" of the child object.
- There are the following "internal triggers for disabling"
    - A child object fires "enabledOut" = "false"
    - A network sensor reports the faulty initialization
    - A child object reports an "internal trigger for disabling"
- At the end of a MOO change in the MOO V (after firing "enabledOut" = "false"), each MIDAS object passes a trigger to the "disable" input of all child objects.