Hibernation of Advanced Railroad Trains (ArrT)

Step Two – Elaborate Visions …... / Leave Hibernation (EViL)

New Concept Paper of SrrTrains v0.01

The „Hibernation of ArrT“ consists of three Steps.

During these three steps the project SrrTrains v0.01 (i.e. the SrrTools software) will not do any external progress, just the internal parts of the software (the SRR Framework and the SMUOS Framework) will be improved:

1. **Step One.....**Relax and Clean up the Project (RaCuP),
2. **Step Two.....**Elaborate Visions..... / Leave Hibernation (EViL)
3. **Step Three.....**...and Communicate them (aCt).

Currently we have developed **Step Two „EviL“**. The SrrTrains v0.01 BIMPF software is currently struggling to develop release „Arimathea“ (expected „not before 2019“), but the part of the SrrTrains v0.01 project that is currently starting to leave the hibernation is not dependent on the BIMPF approach. This part „after the hibernation“ will be something completely new.

The present hibernation report, which is the „New Concept Paper of SrrTrains v0.01“ should help to understand the ideas that were the basis of the SrrTrains v0.01 project with the BIMPF approach.

Chapter „11 The MIDAS Base (MIB)“ is a rather detailed description of the SRR/SMUOS Framework, which was taken from Hibernation Report 001 „Pieta / Rebase to MIB Core“.

Each chapter will be introduced by a short summary of the relevant ideas and then the ideas will be elaborated within the chapter.

Table of Content

1 A Vision of the 3D Web – How To Enter the Internet

A.D. 2002

„What, if we <saved the world> in a distributed, collaborative database of 3D landscape and objects?“

„What's the use case?“

„Use case is any 3D application that needs geo data.“

„Hmmm, we could save fantasy worlds, too, in addition to the <real> world.“

„Well, this could be done similarly as it is done in the WWW with HTML pages.“

„It should use a standardized syntax for transmission of objects and landscape – what about XML?“

„Great! What, if we did not save only the world, but the whole universe?“

This would mean, the VR technology would give us the chance to literally „enter the Internet“, wouldn't it?

The Internet would become an „Enternet“, wouldn't it?

Let's call this distributed, collaborative database „The 3D Web“.

2 Composite Scenes – the Idea DIGITS

A.D. 2002

„If we would like to access the distributed, collaborative database of 3D landscape and objects, wouldn't be the primary keys as follows?

1) A geographical area (let's call it the <virtual roaming area (VRA)>),

2) a level of detail (LoD) to define the highest resolution provided and

3) an indication of which reality we want to receive
(given there are many <realities> stored in the database).“

Let's call this service of the Internet the „Distributed Internet Geographic Information Transmission Service“. This could be a service, which could be used to access basic parts of the „3D Web“.

The basic idea of DIGITS is shown in the following Figure 1:

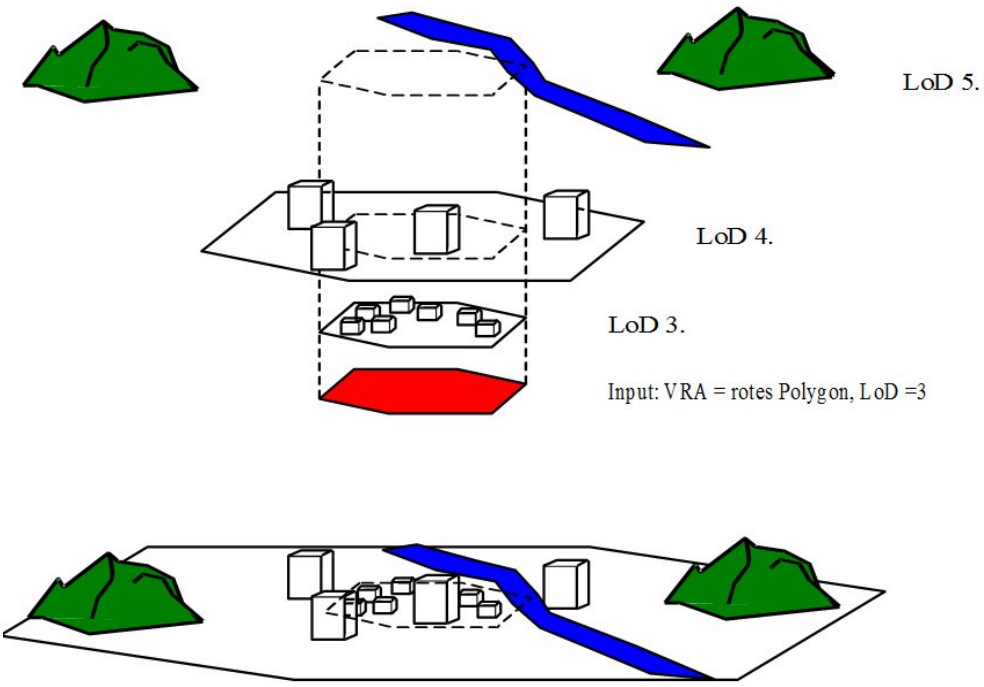


Figure 1: Creating a 3D infrastructure from several LoDs

3 Multiuser Scenes – the Idea SIMUL-RR

A.D. 2007

„What, if we used 3D Multiuser Scenes for training purposes at Railway Operators?“

„Keep it cheap – think about synergies, e.g. with model railroading.“

Singleuser Scenes – merely synonymous for „any kind of application that uses 3D graphics“ - are interesting applications of computers – thinking about myriads of ego shooters and simulation applications.

However, adding the feeling of „working in a team“, i.e. of „collaborating“, adds a lot of fascination to 3D scenes.

Many of such MU scenes would need some „geographic infrastructure“.

In the beginning each scene would implement it's own geographic infrastructure, but eventually all scenes would use some common geographic infrastructure.

Hence the idea of MU scenes would pave the way towards DIGITS and towards the 3D Web.

4 The Network Sensor

„When I asked in early 2009 at the X3D-public mailing list, whether standards existed for 3D Multiuser scenes, they told me about two standards:

1) Distributed Interactive Simulation (DIS)

2) The Network Sensor

Well, later there was another discussion. The following link is a good starting point:
http://www.web3d.org/pipermail/x3d-public_web3d.org/2011-February/001423.html .

Basically, in spring 2009 I decided to use the Network Sensor implementation of the test versions of BS Contact and BS Collaborate from Bitmanagement Ges.m.b.H.

As long as I am only testing my software – not really using it – I think it should be sufficient to use the test versions of BS Contact + BS Collaborate.

However, I did some additional trials with the Octaga Collaboration server. Currently I am not sure whether this software is still available.

5 SrrTrains Community, SrrTrains Core Team, Me

> „Well, when we want to use synergies to keep the software cheap for Railway Operators, then we should install a community that implements the software for free!“

Hmmmmmmmm, it is a good question: „Why am I doing this?“ „Why am I trying to create a framework for Simulated Railroads, based on VRML/X3D and on the Network Sensor?“

Am I really trying to persuade poor hobbyists to sacrifice their ideas and sweat for the good of the industry?

Am I just fascinated by virtual model railroading?

Or is it my faith that a standardized Network Sensor will be necessary for any multiuser scene? Are standards really that important?

For my justification: the project SrrTrains v0.01 is currently resting in „Hibernation Mode“, until it will become more clear to me, whether somebody else is interested in such things, too.

6 Client Based Server Software / The BIMPF Approach

„Using the Network Sensor <as is>, yet enabling MIDAS Objects.“

Chapter ´E.4 A few Words about Synchronization´ explains in a detailed way the steps from using the network sensor over using MIDAS Objects towards using a synchronization with the real reality.

It is clearly explained MIDAS Objects add a level of indirection, because they enable what we call „Client Based Server Software“.

This means each network sensor defines one scene instance being the controller for this network sensor.

Now the actual implementation of the network sensor does not support controller roles, hence we must distribute the change request events to all(!) scene instances, ignoring them in all but one(!) scene instance. This is explained in following Figure 2:

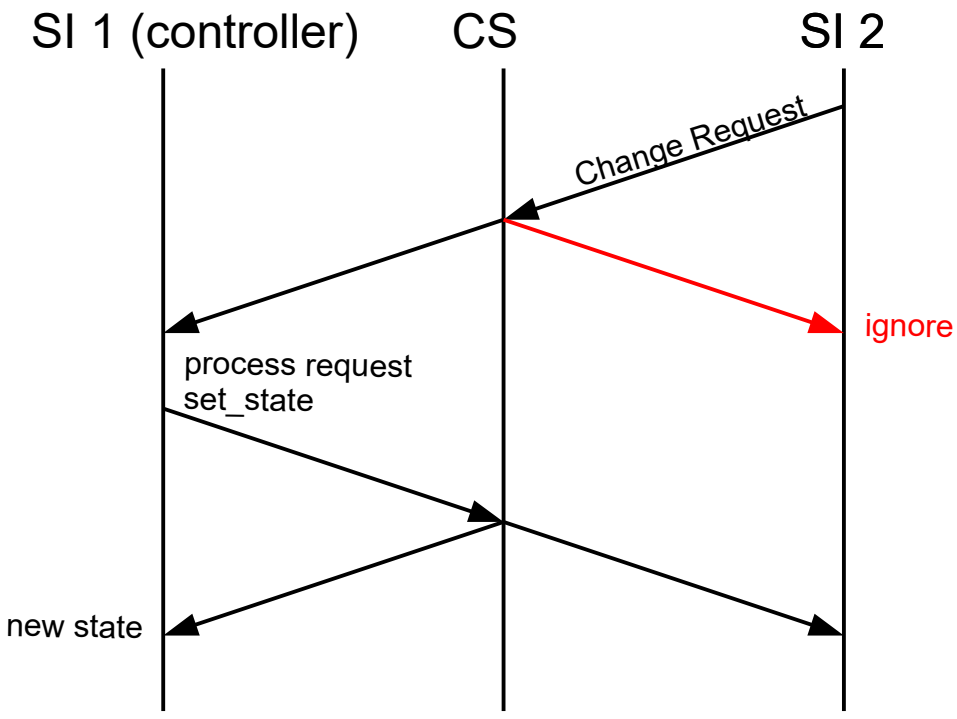


Figure 2: Unnecessary network traffic due to the BIMPF approach

7 The Simple Scene Controller

A.D. 2009:

„Let's start with our project <from the scratch>."

„Let's use one single EventStreamSensor for handling of central information."

„Let's begin with adding a <sessionId> to and removing a <sessionId> from a central <Communication State>"

We did our first experiences with our concept of „Client Based Server Software (CBSS)" implementing and testing the „Simple Scene Controller", which is „the" central part of the SRR Framework.

Our findings were as follows:

If we want our system to be a causal system, then we must establish following laws:

1. We must know, whether we are the first instance that joins a session (then we have to initialize the states of the network sensor – let's call them „global states") or other instances exist already (then we do not initialize the global states, because they are already valid)
2. At any time, only one scene instance is allowed to attain the controller role. Two or more scene instances must never attain the controller role simultaneously
3. We must accept small time gaps, where no controller exists at all and hence events get lost

Following example has got four clients (called „Ssc.Control"), that send events (aka „change requests") to the central server (called „Ssc.CentralSrv"). The central server distributes the global state (aka „Communication State" (commState)) via the CS.

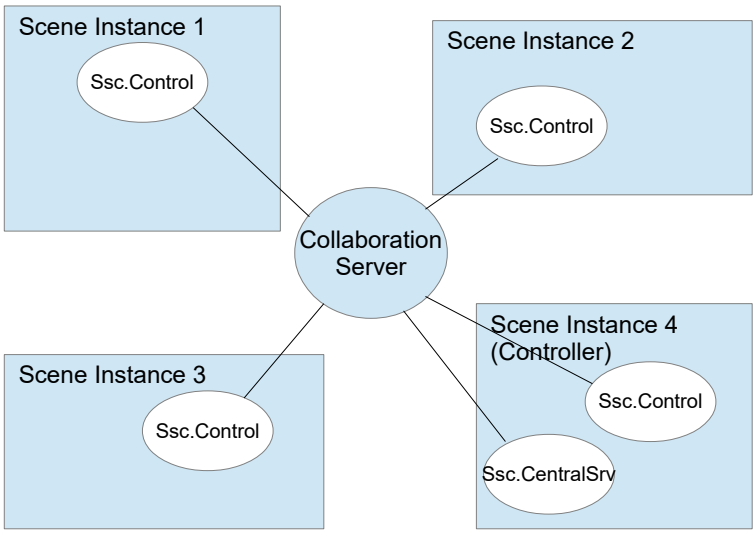


Figure 3: Simple Scene Controller within four scene instances

8 The MMF Paradigm

A.D. 2009 – 2017:

„Let's decompose the model railroad into modules. Each module could be authored by a different person.“

„OK, then let's assign each object – each vehicle, each house, each car and so on – to a module. This would make it possible to agglomerate properties of the objects into properties of whole modules:

– controller roles of the objects,

– activity of the objects.“

„We could define a module as <inactive>, when the user is not near to it?“

„Let's define an abstract <module activity>. Each object on the module is informed about the <module activity state> and can do its own decision about how to react.“

„The module author can decide, when his module gets active and when it gets inactive.“

„Let's define a <module controller role (MOC)> and define: the controller roles of all objects (OBCOs) will <follow> the module controller role (MOC) of the parent module.“

Well, basically each object should be attached to a module. Reason is each module spans a local coordinate system and hence rendering of objects makes perfect sense relative to a module.

Rendering models relative to a – philosophically problematic – world coordinate system does not really make sense. We define some „invisible infrastructure“ of the scene, which can be related to the „World Coordinate System“, and call it „the Frame“.

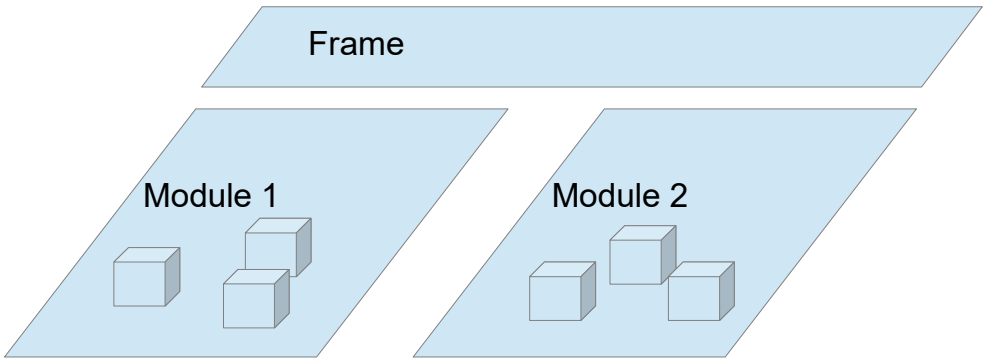


Figure 4: Objects attached to Modules (Bound Objects - BOBs)

Almost from the beginning of the project there was one object that needed not be attached to a module. This object was the „Avatar Container“. It was possible to hold Avatar Containers within the Frame. Later we called such objects the „Astral Objects“.

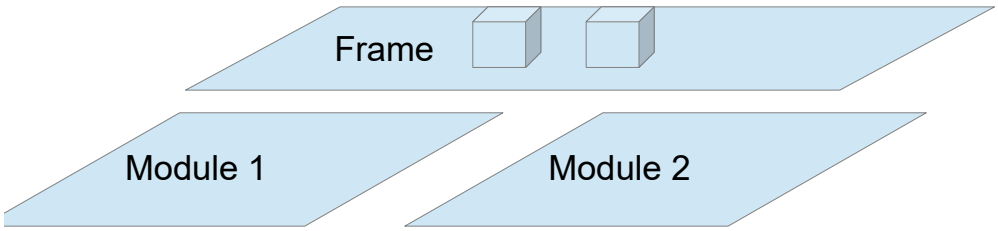


Figure 5: Objects not attached to any module (Astral Objects - AOBs)

Rendering astral objects does not make sense.

Now, with the upcoming Step 0033.11 of the software, we will have what we call „Unbound Objects“ (but not before 2019).

Unbound objects are usually attached to a module – as bound objects are – and can be rendered therefore.

However, unbound objects can change the module they are attached to (we call this a „handover“) and they can even act as astral objects temporarily – e.g., if the module they are attached to, becomes unloaded by some reason.

Changing from a module to the frame is called a „DeAttachment“, changing back to a module is called an „Attachment“.

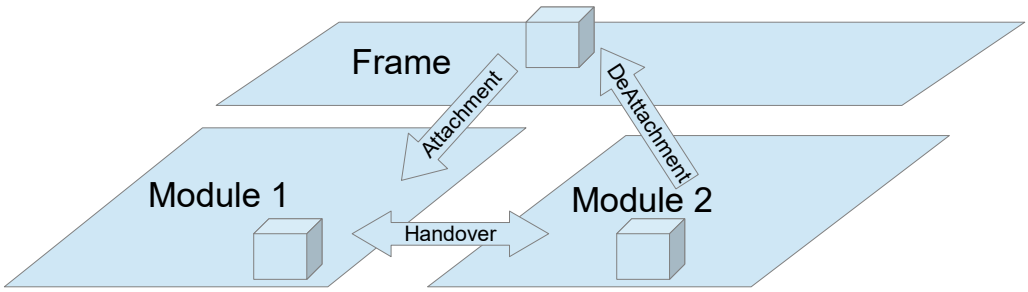


Figure 6: Unbound object (UBO) changing being attached to a module

9 MAM, OBCO and MIDAS Objects

A.D. 2009 – 2017:

„Well, let's make the <Simple Scene Controller> maintain the <Activity State> of the modules, OK?“

„Good idea! But don't forget: the activity of a module is a local category, but not a global. Since the activity of a module might depend on the position and orientation of the local user, then the module activity is a matrix of boolean values – let's call it the <Module Activity Matrix> (MAM).“

„OK, let's additionally define, only an active instance of a module can attain the MOC role. If all instances of a module are inactive, then the module has not got a MOC. In this case the objects in the module don't have object controllers (OBCOs), either.“

„Let's define the objects that control the simulation are called MIDAS objects, i.e. <Multiuser Interactivity Driven Animation and Simulation Objects>“

The activity states and the MOC roles of all modules are maintained by the Simple Scene Controller within the Communication State.

If the MOC role switches from one instance of a module to another instance, then this is done within one and the same update of the Communication State. This is shown in Figure 7.

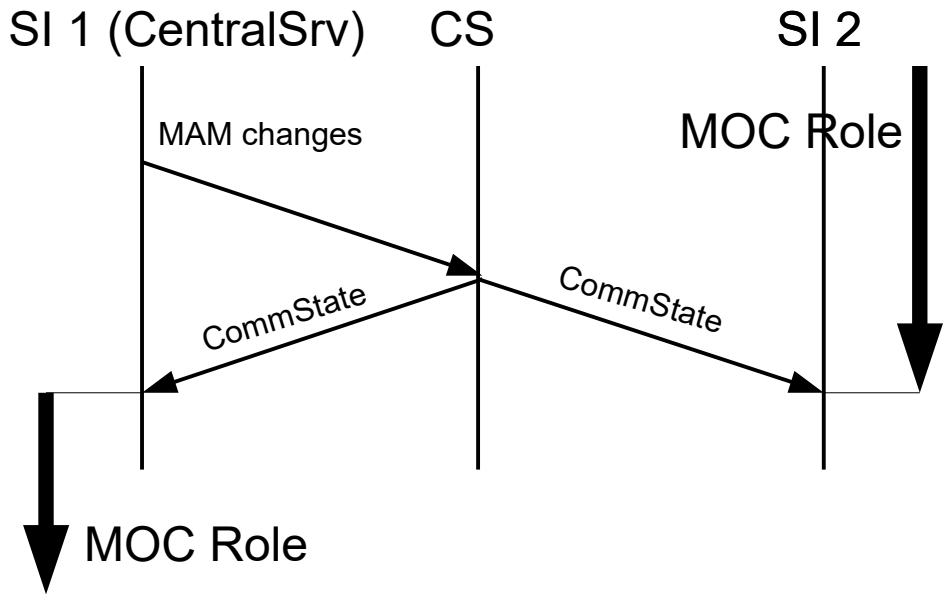


Figure 7: Changing the MOC Role of a module by CentralSrv

The activity state of a module instance is indicated by one of the characters 'o', '-', '+' or '*'.

- 'o' means the module instance has not yet been loaded nor initialized
- '-' means the module instance is inactive
- '+' means the module instance is active (without having got the MOC role)
- '*' means the module instance is active and it has got the MOC role

Figure 8 displays an example with four scene instances (sessionIds) and four registered modules, each scene instance containing an instance of each module.

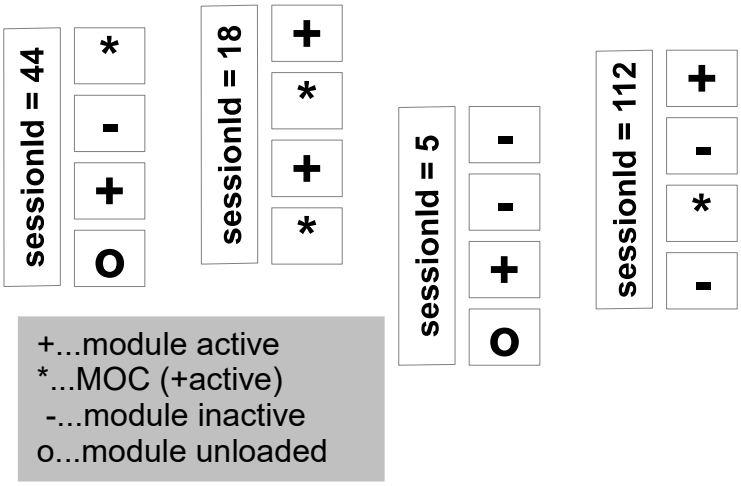


Figure 8: Example of a module activity matrix (MAM)

In chapter „7 The Simple Scene Controller„ we have explained, why the Simple Scene Controller contains a Network Sensor to maintain the „Communication State“. Now we saw the Communication State contains also the MAM, to distribute the activity state of all modules.

But still the „Module Activity“ is something abstract. We do not know the purpose of the MAM.

We must explain the purpose of the MIDAS Objects, if we want to understand the purpose of the MAM.

MIDAS Objects are embedded within models to instrument the behaviour of the models. Imagine a car, then you will need a steering and a motor and you will need some user interface to influence the steering and the motor.

MIDAS Objects can be compared to the steering and to the motor of a car. Both are invisible (they are not rendered), but they influence the behaviour of the car.

Now, in the case of multiuser scenes, the steering and the motor must be synchronized among all scene instances. This is done using Network Sensors.

Also the MIDAS Objects – not only the Simple Scene Controller – are built according to the paradigm of „Client Based Server Software“.

This means in particular that each MIDAS Object defines a controller role, the so-called „Object Controller (OBCO)“.

We decided to let the OBCO Roles follow the MOC Roles. Hence the OBCO roles of all MIDAS Objects that are attached to the same module, follow the MOC Role of that module.

Each MIDAS Object contains a little state machine, the „OBCO State Machine (OSM)“.

This state machine is attached to the module together with the MIDAS Object and it receives commands from the Module Coordinator.

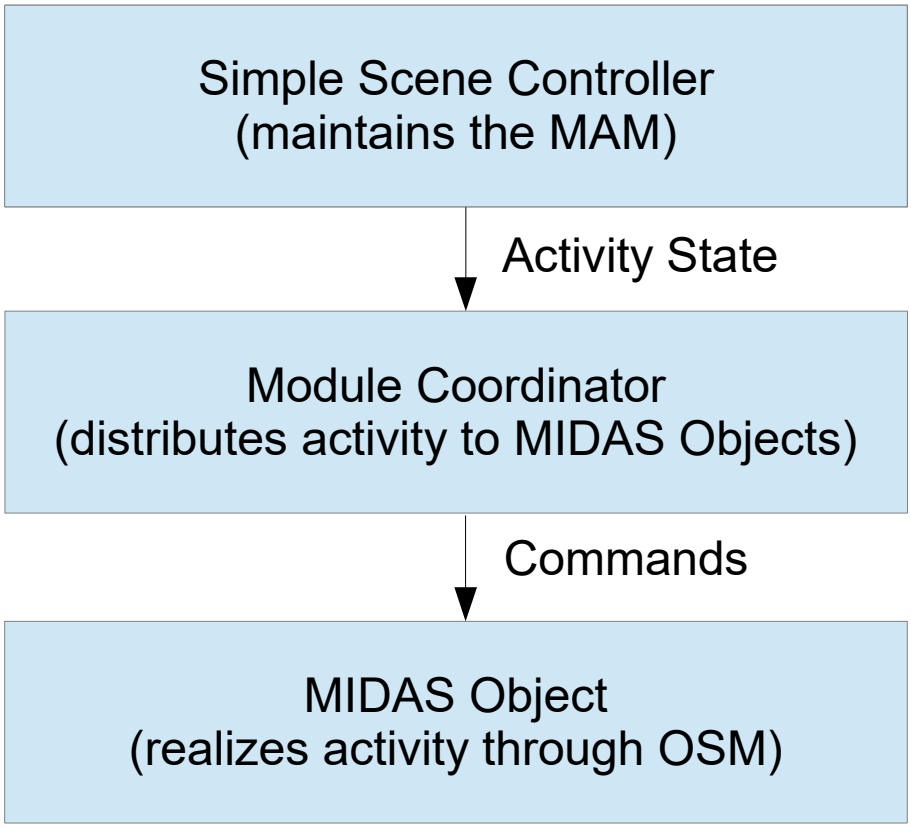


Figure 9: Distribution of the activity state towards MIDAS Objects

The comands ***takeMOC***, ***grantMOC***, ***activate*** and ***deactivate*** need not be explained.

The command ***disable*** is sent, when a complete module is disabled and hence all MIDAS Objects need to be disabled.

The command ***sessionIds*** is a list of all sessionIds, in which this module is active. This list is needed by some MIDAS Objects.

10 Unbound Models and Handover

A.D. 2009 – 2017:

„Till now, we identified objects by <moduleName> + <objId>. Now, when a model can change the module it is attached to, this is not possible any more“

„Hmmm, let's define <Universal Object Classes> (UOCs). Each unbound object shall be identified by <uocName> + <objId>“

„Good idea. We have the possibility to extend the SSC by SSC Extensions. Let's give any SSC Extension the choice to implement one or more UOCs“

Unbound models still need to be implemented.

Currently we are planning to have it finished by 2019.

Handover will be implemented even later.

11 The MIDAS Base (MIB)

A.D. 2009 – 2017:

„Let's implement some base functions to ease the programming of MIDAS Objects. Let's call it the <MIDAS Base> (MIB)“

11.1 Simple Multiuser Scenes (SMS) – An Introduction

At the very beginning of the SrrTrains v0.01 project we decided to use Web3D Browsers that comply to the X3D standard.

The X3D standard is a very general ISO standard that can be used for virtually any 3D scene in the World Wide Web.

However, our aim was something more special: we were aiming at what we called "Simple Multiuser Scenes (SMS)".

Any author should be able to easily implement an SMS, based on Web3D Browsers.

Now it turned out that many of the work that had to be done to implement an SMS, would always be the same, at least very similar, for each and every SMS again and again.

This led to the goal to implement a general framework – as general as to be useful for any SMS – but not as general as a Web3D Browser.

The extensibility of X3D turned out to be a very useful concept for this purpose.

This can be shown in a layered view.

The lower a layer the more general it is and the higher a layer the more specialized.

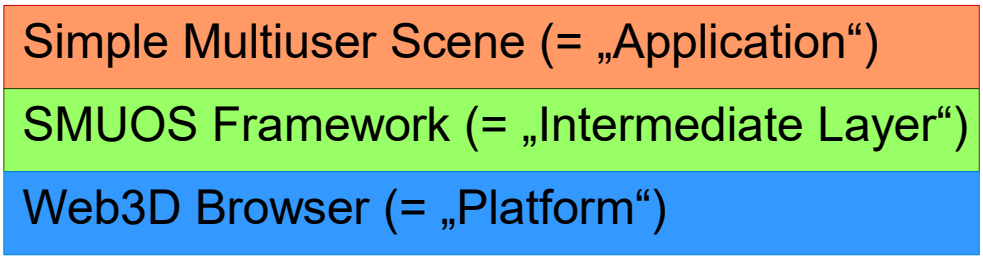


Figure 10: A layered view of a Simple Multiuser Scene (SMS)

11.2 The SMUOS Framework – Overview

The SMUOS Framework (Simple Multiuser Online Scenes Framework), which is actually a part of the SRR Framework, requires from the scene to follow **what we call the "MMF Paradigm"**.

This just means the authors have to decompose the scenes into ***m***odels, ***m***odules and ***f***rames.

Figure 11 tries to explain the relations among the parts of the SMUOS Framework and the models, modules and frame of a scene.

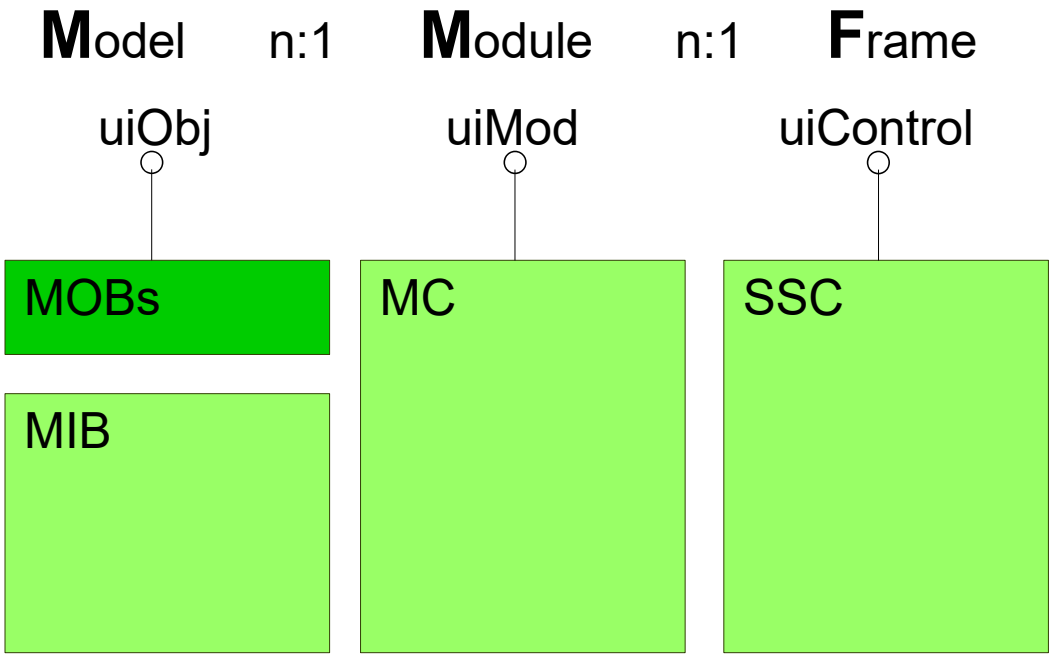


Figure 11: Parts of the SMUOS Framework (SSC, MC, MIB) and MIDAS Objects (MOBs)

The light green parts, MIB, MC and SSC, are actually parts of the SMUOS Framework, where the dark green part (the MOBs) are not actually parts of the SMUOS Framework, but they are closely related to the SMUOS Framework.

Each of the three light green parts has got an own core prototype during step 0033.10 – as described in chapter „11.2.3 Three Parts of the SMUOS Framework – Decomposition„ - but first let's dig a little bit into the basics:

11.2.1 Are You an Author or Are You A Programmer

Well, when we talk about computer graphics, then the walls between programmers and authors will probably melt down.

Authors can only build sophisticated features of their scenes, when they are able to do this or that coding gimmick.

Programmers can only test their software, when they are able to build scenes specific for this or that purpose.

On the other hand, we make this difference between "authors" and "programmers" to indicate the parts of the software that are more graphically involved (those parts are created by what we call "authors") and the parts of the software that are more programmatically involved (those parts are created by what we call "programmers").

> *Definition: We arbitrarily set a border between authors and programmers, which we call the **user interfaces**.*

1. The **user interface uiObj** is the interface between MIDAS Object and Model, where a programmer provides a MIDAS Object and an author uses the MIDAS Object to instrument a model.
2. The **user interface uiMod** is the interface between Module Coordinator and Module, where a programmer provides the module coordinator and an author uses the Module Coordinator to instrument a module.
3. The **user interface uiControl** is the interface between Simple Scene Controller and Frame, where a programmer provides the Simple Scene Controller and an author uses the Simple Scene Controller to instrument a frame.

11.2.2 External Interfaces

In chapter „11.2.1 Are You an Author or Are You A Programmer„ we explained the user interfaces.

Well, every user interface is an **external interface**, but not every external interface is a user interface.

> *Definition: External interfaces are well defined interfaces of the SMUOS Framework that can be used by programmers or by authors to extend or to use the SMUOS Framework.*

Interfaces that can only be used internally to the SMUOS Framework or interfaces that can be used externally but are not well defined, are not denoted "external" interfaces.

So the MIDAS Base provides an external interface to the programmer, who wants to build a MIDAS Object. **This interface is called eiMib**.

A programmer of a MIDAS Object could omit the usage of the eiMib interface and directly use the MC and SSC via the **external interfaces eiMod and eiControl**, respectively.

These considerations are shown in Figure 12.

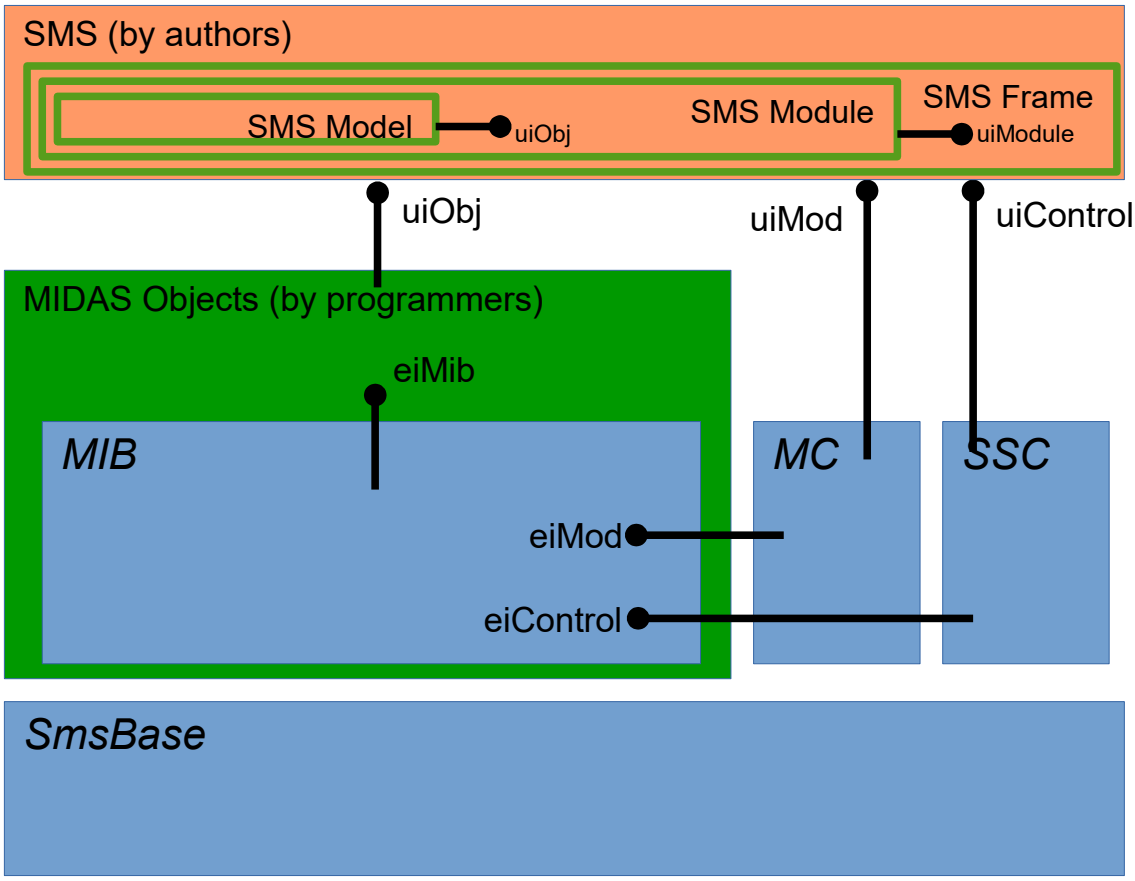


Figure 12: External interfaces of the SMUOS Framework

In Figure 12 we do not only see the external interfaces of the SMUOS Framework, but we also see what we call **minimum interfaces**.

Some parameters must be transported from the SSC to the MC via the frame and the module, or from the MC to the MIDAS Objects via the module and the model.

Hence the SMUOS Framework requires minimum sets of fields that must be present at the external interface of any SMS Module (**uiModule**) and of any SMS Model (**uiObj**), respectively.

11.2.3 Three Parts of the SMUOS Framework – Decomposition

11.2.3.1 MIDAS Objects (MOBs) and MIB – What's Their Purpose?

The magic of **MIDAS objects (MOBs)** is the "magic of the intermediate layer".

What does this mean? Well, when we are using the Network Sensor, then we can build virtually anything. Anything that needs a shared state in a multiuser scene can be built on the Network Sensor. So why do we need MOBs?

Well, if a scene author wants to build a car on the Network Sensor, than he needs to implement a steering, a motor, some shared state for the doors to get open or closed and so on, that's all.

Unfortunately **each and every scene author**, who uses the Network Sensor to build a car, **has to do the same work again and again**: building steerings, motors and other specialized mechanisms from the general Network Sensor.

The concept of the MOBs allows to **build specialized mechanisms that can be re-used again and again**, thus saving tons of effort.

Hence it must be clear that the MOBs cannot be parts of the SMUOS Framework, because the SMUOS Framework is a rather general appliance and the MOBs are rather specialized appliances.

Nevertheless, we have identified some common properties that apply to all MOBs. We grouped these properties according to three types of MOBs and built the base software for MOBs – i.e. the **MIDAS Base (MIB)**. This is actually a part of the SMUOS Framework.

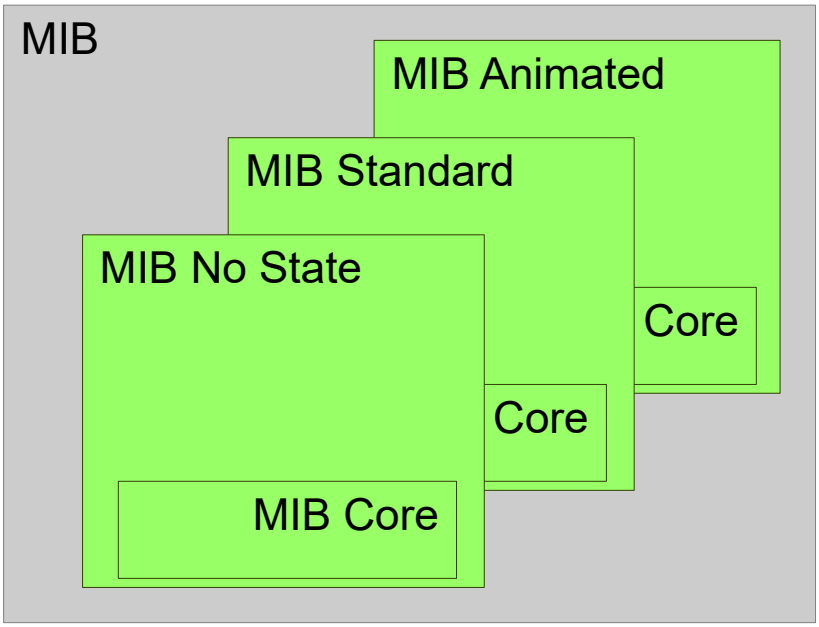


Figure 13: MIDAS Base (MIB) for three types of MIDAS Objects (MOBs)

11.2.3.2 Module Coordinator (MC) – What's its Purpose?

One basic concept of the SrrTrains v0.01 project is to decompose the landscape into what we call "modules".

The first idea was to give more than one scene author the possibility to work together on one landscape (on what we call "layout").

Additionally, all MOBs within one module might share common properties, e.g. we could "deactivate" a module, when the user would not be interested in this module (e.g. he would be roaming far away from the module and he would not look at the module).

So the basic duty of the **Module Coordinator (MC)** is to coordinate all MOBs of one module.

Later we found that some specialized MIDAS Objects could require more functionality from the MC than that, which is present in the basic MC (**MC Base**).

This led to the possibility to extend the MC by your own extension (**MC Extension**).

Some basic properties that are of common interest for all MC Extensions have been outsourced to the **MC Core** in step 0033.10 (Release "Pieta").

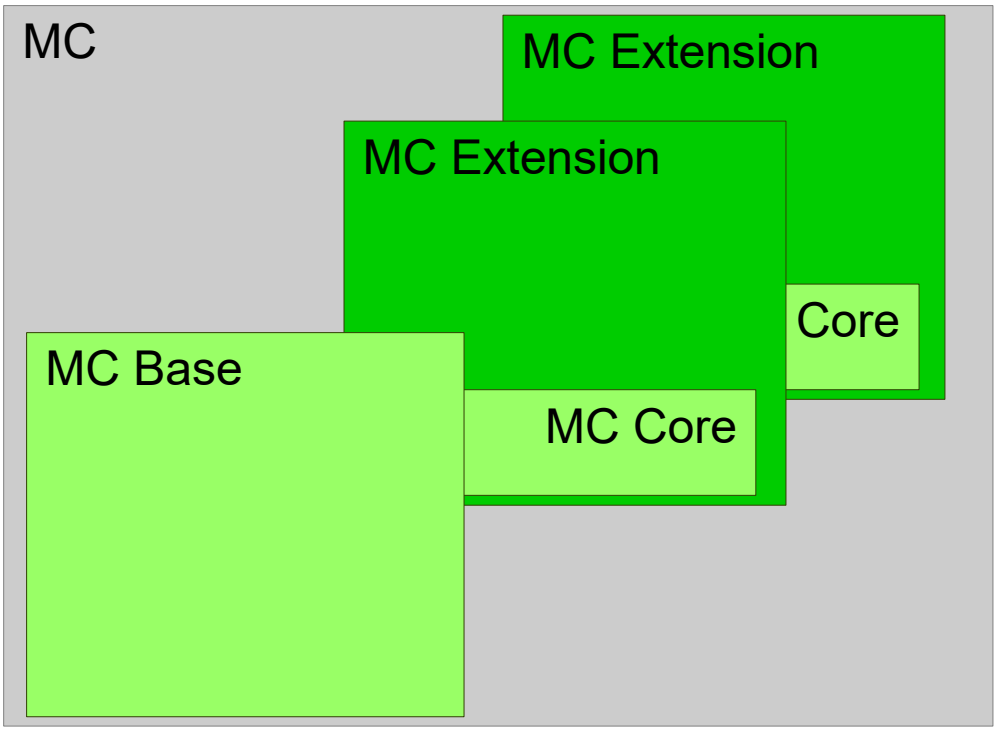


Figure 14: MC Base, MC Core and 3rd Party MC Extensions

The dark green parts are provided by external programmers / projects (3rd parties), where the light green parts are actually parts of the SMUOS Framework.

11.2.3.3 Simple Scene Controller (SSC) – What's its Purpose?

After discussing the smallest parts of the scene – (M)odels using MIDAS Objects using the MIB – and the second-smallest parts of the scene – (M)odules using the Module Coordinator – it is now time to discuss the (F)rame.

The (F)rame is the part of the scene that provides all common and basic functionalities. It supports (M)odels and (M)odules in presenting a scene to the user.

Hence we say the user can **inhabit** a virtual scene (SMS).

The frame needs a **central access point** to access the functions of the SMUOS Framework and the Module Coordinators and the MIDAS Objects need some **central intelligence**.

This central access point and intelligence are provided by the **Simple Scene Controller (SSC)**.

Some MIDAS Objects and some MC Extensions might need additional functions that are not provided by the **SSC Base**.

Hence we added the possibility to implement your own **SSC Extensions**.

In step 0033.10 (Release "Pieta") we outsourced some common functions that are common to all SSC Extensions into the **SSC Core**.

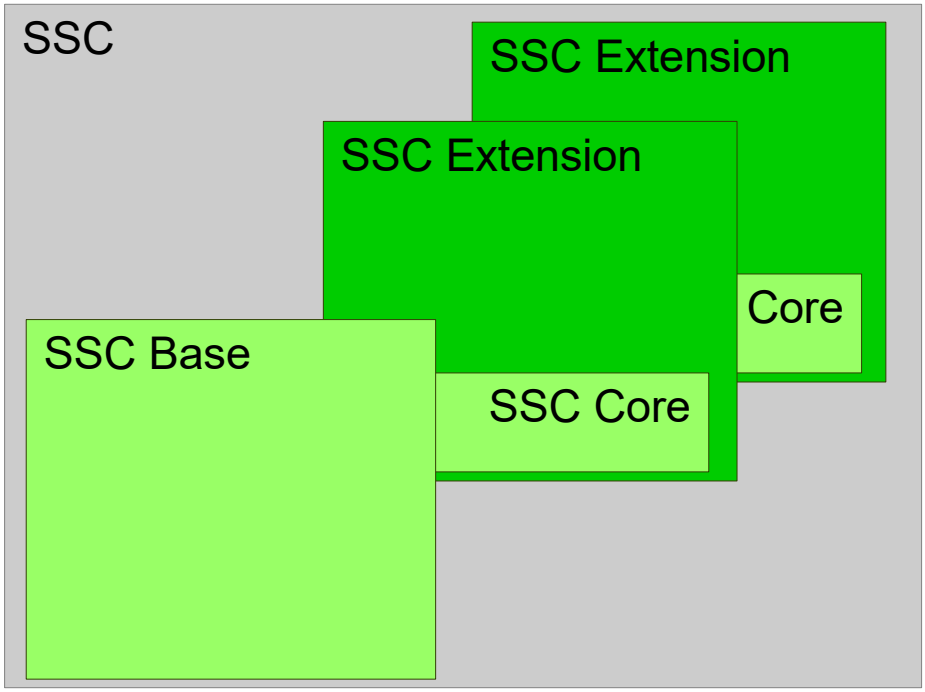


Figure 15: SSC Base, SSC Core and 3rd Party SSC Extensions

The dark green parts are provided by external programmers / projects (3rd parties), where the light green parts are actually parts of the SMUOS Framework.

11.3 The Simple Scene Controller

Chapter „11.2.3.3 Simple Scene Controller (SSC) – What's its Purpose?„ introduced the Simple Scene Controller (SSC) in a short and concise way.

The present chapter describes the SSC in a more detailed way, in particular we focus on the reader, who wants to implement an SSC Extension.

11.3.1 Purpose of the Simple Scene Controller

The SSC tries to be a **central access point within each scene instance (SI 1, SI 2, ...)**.

That means, all aspects of the SMUOS Framework that are relevant to the whole simulation, can be accessed via the **user interface uiControl**.

The SSC needs to maintain a global state (i.e. a set of values that are identical to all scene instances). Regarding this global state we distinguish the common **Communication State (commState)**, which is maintained by the SSC Base, and the specific **global state of each SSC Extension**.

Please find an overview depicted in Figure 16.

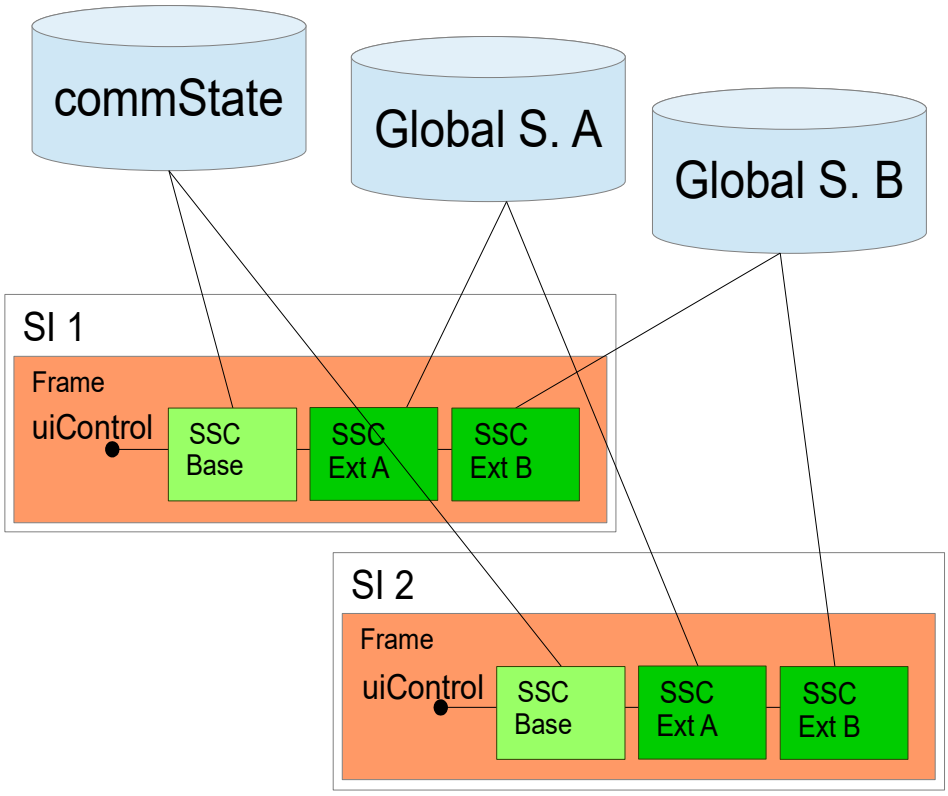


Figure 16: Global commState and global states of all SSC Extensions

The **initialization of the SSC within one scene instance (SI)** is started by the user of the SSC (i.e. by the frame author), when he sends the event "init" via the uiControl Interface to the SSC Base.

Then the SSC Base and all dependent SSC Extensions are initialized, i.e. they change from Mode of Operation (MOO) "LOADED" to Mode of Operation (MOO) I "initialized".

This is not enough to issue the Common Parameters (commParam), because the global state "Communication State (commState)" must be handled in advance. This initial handling of the global state is called "activation".

After the commParam have been issued, then the SSC Extensions are activated, too.

Please find an illustration in Figure 17 and in the following listing.

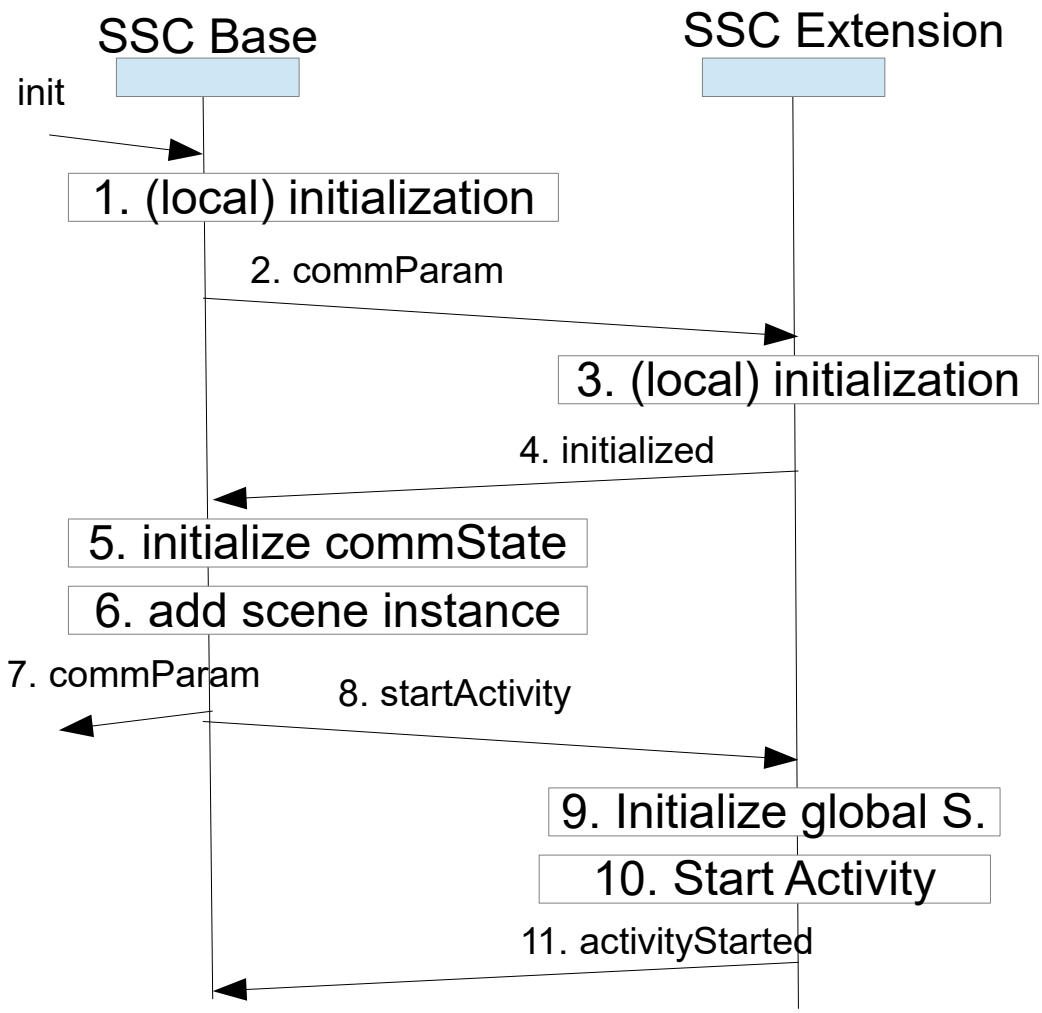


Figure 17: SSC initialization and start activity – information flow

1. Local Initialization of the SSC Base
   The frame of the scene triggers the initialization of the SSC with the help of the field "init" at the uiControl interface.
   Only the most important fields of the Common Parameters (commParam) are initialized now, just to allow all SSC Extensions to be initialized locally
2. Forwarding of commParam to all dependent SSC Extensions
   All dependent SSC Extensions are triggered to get initialized by the commParam
3. Local Initialization of all dependent SSC Extensions
   Each SSC Extension (exactly the SSC Core in each SSC Extension)
   - triggers and waits the local initialization of all mandatory dependent SSC Extensions
   - triggers and waits the local initialization of all optional dependent SSC Extensions
   - triggers and waits the initialization of all SSC Dispatchers of the SSC Extension
   - triggers and waits the call back procedure "initialization" (here the author of the SSC Extension can include self-written code)
   - reports "initialized" in case of successful local initialization (see step 4.)
   - triggers the initialization of all network sensors of the SSC Extension
4. Local Initialization of all dependent SSC Extensions is finished
5. Initializing the commState
   If the scene instance is the first instance of the multiuser session, then the commState will be explicitely initialized to "empty"
6. Add Scene Instance to commState
   The scene instance sends an "Access Request" to the server part of the SSC and hence the scene instance will be added to the commState
7. Now the Common Parameters are valid and the flag "iAmActive" in the commParam is set to TRUE. The flag "iAmActive" of the Common Parameter **Extensions** is still FALSE. However, the commParam can be used by the scene now and hence are published by the SSC via the uiControl interface.
8. If the scene instance is the first of the multiuser session, then the SSC Extensions are triggered with the field "initializeStateAndStartActivity", otherwise they are triggered with "startActivity"
9. Each SSC Extension initializes its global state (if it was triggered with "initializeStateAndStartActivity" – otherwise the state is already valid)
10. Now the local copy of the global state is valid and the SSC Extension can "start Activity". The meaning of "start Activity" may vary from SSC Extension to SSC Extension
11. The SSC Extension reports that activity has been started, the "iAmActive" flag in the Common Parameter Extension is set to TRUE.

11.3.2 The Common Parameters (commParam)

The basic idea of the Common Parameters is to have a <Script> node within the SSC that holds parameters for the whole scene instance, i.e. parameters that are "common" to the whole scene instance.

Now when we defined that the SSC Base may be extended by SSC Extensions, then the idea was obvious to add "Common Parameter Extensions" to the Common Parameters, one for each SSC Extension.

The Common Parameters got a field "extensions (MFNode)", which pointed to the Common Parameter Extensions.

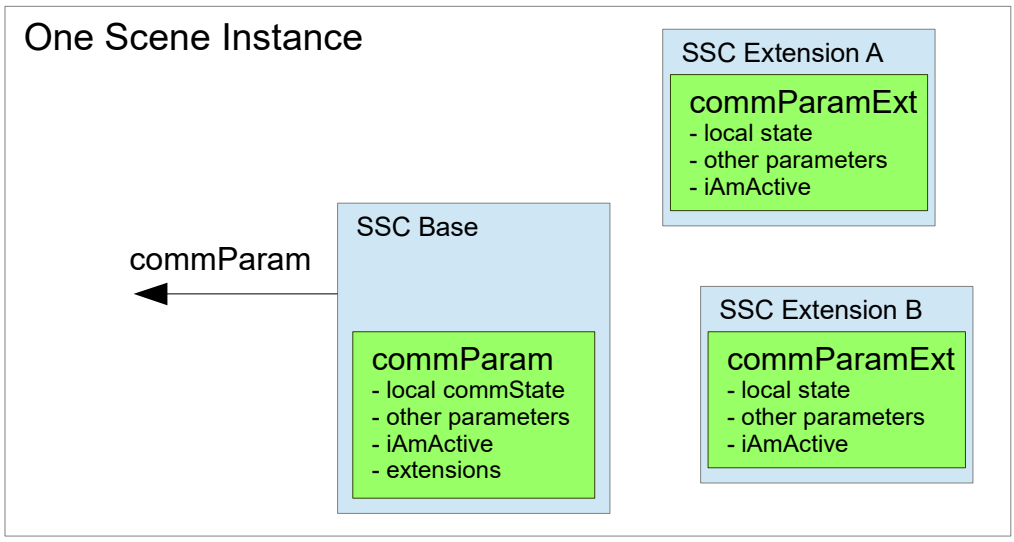


Figure 18: The concept of commParam and commParamExt

Each of those script nodes would hold a local copy of the global state of the part of the SSC (i.e. of the commState for SSC Base and of the global state for each SSC Extension).

The "iAmActive" flag, which we talked about in the above chapter, would also be located within those script nodes.

Last but not least, the commParam would point to all commParamExt and each commParamExt would point to the commParam.

When SSC Base publishes the "commParam" reference, then it guarantees that all mandatory dependent SSC Extensions are successfully initialized, that all optional dependent SSC Extensions are successfully or unsuccessfully initialized and that the field "extensions" points to all successfully initialized dependent SSC Extensions, but not to any unsuccessfully initialized dependent SSC Extension.

11.3.3 Interfaces of the Simple Scene Controller

This chapter tries to give some hints for those, who like to implement their own SSC Extension.

Following figure should give a short and concise introduction:

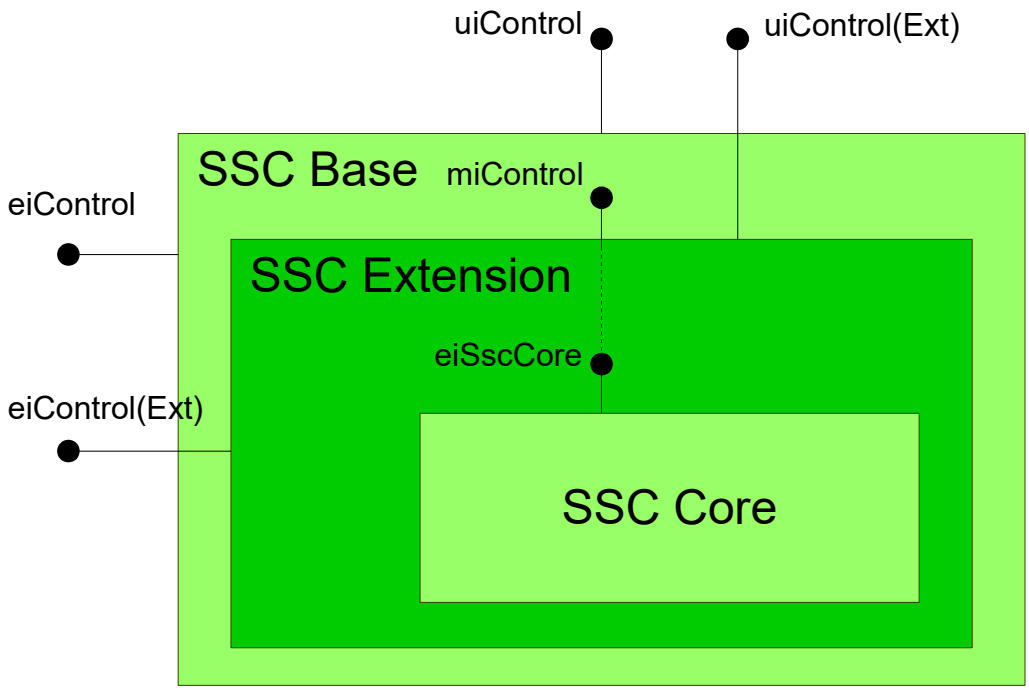


Figure 19: Considerations for the implementation of an SSC Extension

An SSC Extension

- **must** provide the **minimum interface miControl** to the SSC Base (or to the parent)
- **may** use the **external interface eiSscCore** of the SSC Core to ease the implementation
- **may** provide a **user interface uiControl(Ext)** to the frame.
- **may** provide an **external interface eiControl(Ext)** to its MC Extension(s) and/or to its MIDAS Objects

11.4 The Module Coordinator

Chapter „11.2.3.2 Module Coordinator (MC) – What's its Purpose?„ introduced the Module Coordinator (MC) in a short and concise way.

The present chapter describes the MC in a more detailed way, in particular we focus on the reader, who wants to implement an MC Extension.

11.4.1 Purpose of the Module Coordinator

The module coordinator

- coordinates all MIDAS Objects within one module within one scene instance (SI)
- enables access to the global state(s) of the SSC (Extensions) by MIDAS Objects

The first task will become more understandable in the next chapters, the second task is depicted in Figure 20 at an example, where SMUOS Extension A consists of an SSC Extension A and SMUOS Extension B consists of an SSC Extension B and an MC Extension B.

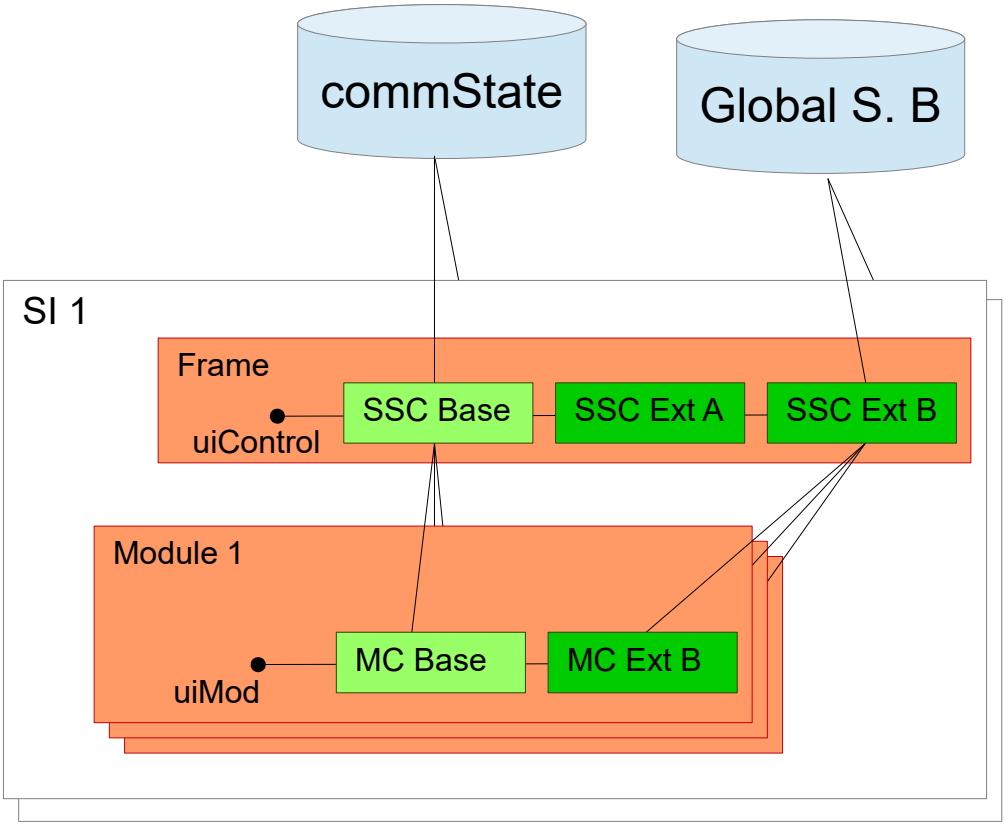


Figure 20: Access to SSC global state(s) via the SSC (Extensions)

The **initialization of the Module Coordinator (MC) within one instance of a module** is started by the user of the MC (i.e. by the module author), when he sends the event "commParam" via the uiMod interface to the MC Base.

Then the MC Base and all MC Extensions change their Mode of Operation (MOO) from MOO "LOADED" to MOO I "initialized".

This is not enough to issue the module parameters (modParam), because the MC Base must be **attached to the SSC Base** in advance, in order to **access the Module Activity Matrix (MAM)**, which is a part of the commState. Now the MC Base changes to MOO II "attached".

After the modParam have been issued, the MC Extensions are attached to their SSC Extensions, too.

Please find a detailed explanation in Figure 21 and in the following listing:

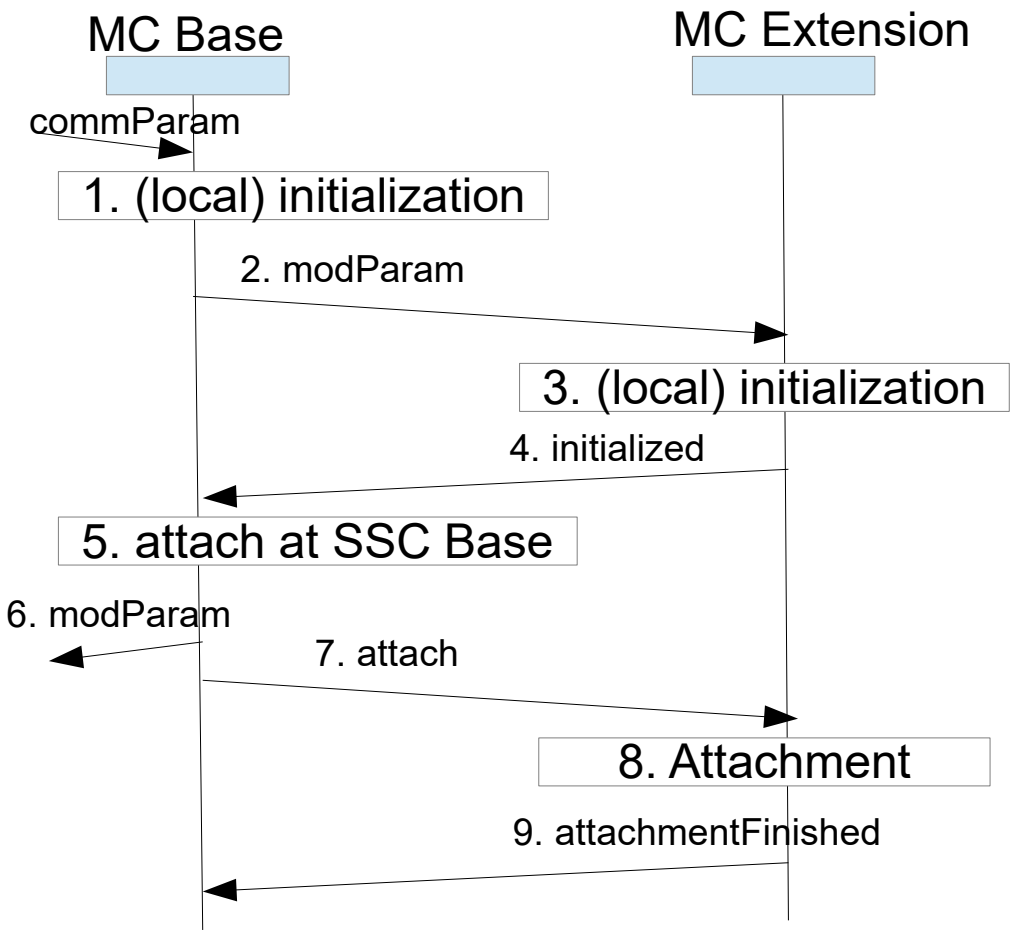


Figure 21: MC initialization and attachment - information flow

1. Local Initialization of the MC Base
   The instance of the module triggers the initialization of the MC with the help of the field "commParam" at the uiMod interface.
   Only the most important fields of the Module Parameters (modParam) are initialized now, just to allow all MC Extensions to be initialized locally
2. Forwarding of modParam to all dependent MC Extensions
   All dependent MC Extensions are triggered to get initialized by the modParam
3. Local Initialization of all dependent MC Extensions
   Each MC Extension (exactly the MC Core in each MC Extension)
   - triggers and waits the local initialization of all mandatory dependent MC Extensions
   - triggers and waits the local initialization of all optional dependent MC Extensions
   - triggers and waits the call back procedure "initialization" (here the author of the MC Extension can include self-written code)
   - reports "initialized" in case of successful local initialization (see step 4.)
4. Local Initialization of all dependent MC Extensions is finished
5. Attach the MC Base to the MAM (SSC Base)
   The MC Base sends an "Announcement Request" to the SSC Base and requests to be attached to the commState. Hence the module gets a "moduleIx", and if it is not yet registered, then it will be implicitly registered
6. Now the Module Parameters are valid and the index "moduleIx" in the modParam is set to the valid value >= 0. The index "moduleIx" of the Module Parameter Extensions is still < 0. However, the modParam can be used by the scene now and hence are published by the MC via the uiMod interface.
7. The MC Extensions are triggered with the field "attach"
8. Now each MC Extension can perform the "Attachment". The meaning of "Attachment" may vary from MC Extension to MC Extension
9. The MC Extension reports that attachment has been finished, the "moduleIx" index in the Module Parameter Extension is set to the correct value >= 0.

11.4.2 The Module Parameters (modParam)

The basic idea of the Module Parameters is to have a <Script> node within the MC that holds parameters for the whole instance of the module, i.e. parameters that are "common" to the whole instance of the module and its MIDAS Objects.

Now when we defined that the MC Base may be extended by MC Extensions, then the idea was obvious to add "Module Parameter Extensions" to the Module Parameters, one for each MC Extension.

The Module Parameters got a field "extensions (MFNode)", which pointed to the Module Parameter Extensions.

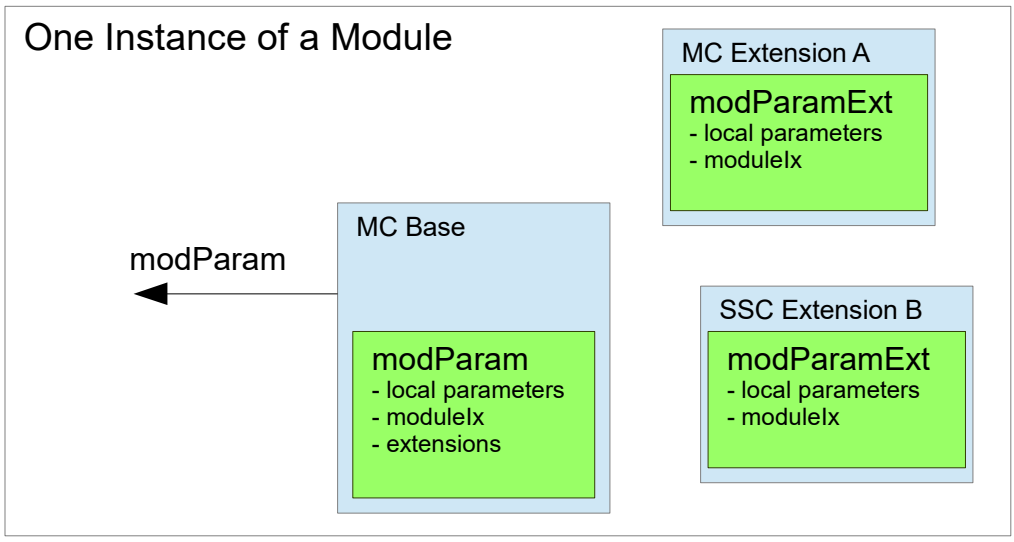


Figure 22: The concept of commParam and commParamExt

The "moduleIx" index, which we talked about in the above chapter, would also be located within those script nodes.

Last but not least, the modParam would point to all modParamExt and each modParamExt would point to the modParam.

When MC Base publishes the "modParam" reference, then it guarantees that all mandatory dependent MC Extensions are successfully initialized, that all optional dependent MC Extensions are successfully or unsuccessfully initialized and that the field "extensions" points to all successfully initialized dependent MC Extensions, but not to any unsuccessfully initialized dependent MC Extension.

11.4.3 Interfaces of the Module Coordinator

This chapter tries to give some hints for those, who like to implement their own MC Extension.

Following figure should give a short and concise introduction:

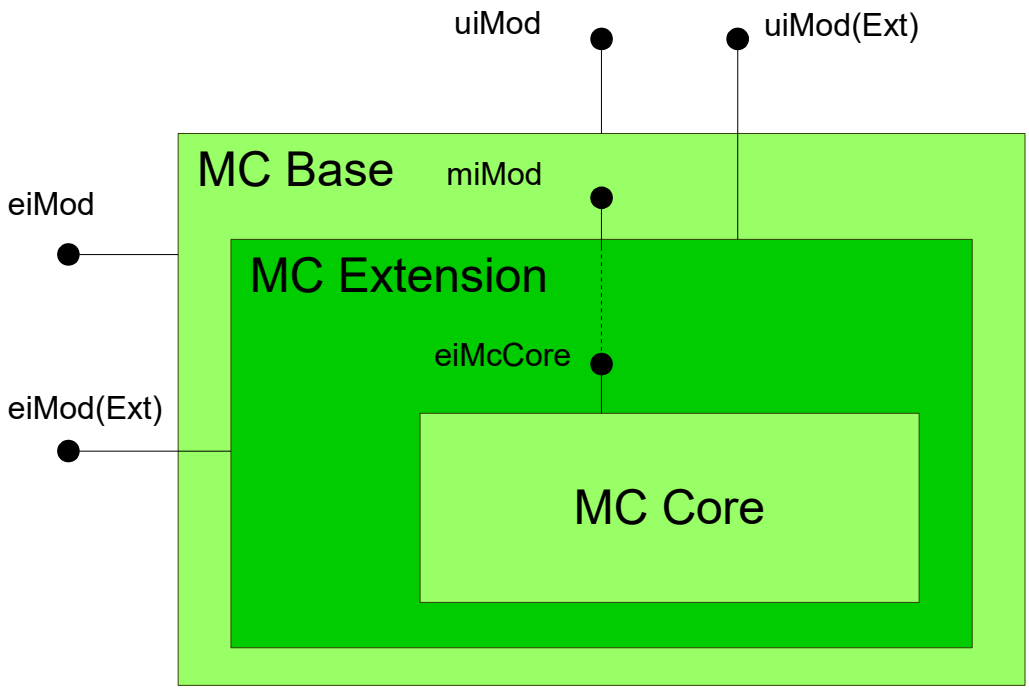


Figure 23: Considerations for the implementation of an MC Extension

An MC Extension

- **must** provide the **minimum interface miMod** to the MC Base (or to the parent)
- **may** use the **external interface eiMcCore** of the MC Core to ease the implementation
- **may** provide a **user interface uiMod(Ext)** to the module.
- **may** provide an **external interface eiMod(Ext)** to its MIDAS Objects

11.5 MIDAS Objects and the MIDAS Base

Chapter „11.2.3.1 MIDAS Objects (MOBs) and MIB – What's Their Purpose?„ introduced the MIDAS Objects (MOBs) and the MIDAS Base (MIB) in a short and concise way.

The present chapter describes these concepts in a more detailed way, in particular we focus on the reader, who wants to implement an own MIDAS Object.

11.5.1 Purpose of MIDAS Objects (MOBs)

Short and sweet, it's the purpose of MOBs to **instrument models**.

Let's have a look at an example in the official demo layout of the SIMULRR project:

The station house is a model, which is located in the module "City". The extended object ID (extObjId) of the station house is **City-StationHouse**.

This is a so-called **bound model** (this term will be explained later).

Now the station house features a door, which can be open, closed, locked or unlocked. Therefore the station house contains two MIDAS Objects, the MoosSwitchA object and the MoosLockB object. The lock is contained in the switch to be able to lock or unlock the switch.

Hence the switch and the lock got following extended object IDs:

1. door switch: extObjId = **City-StationHouse.DoorSwitch**
2. door lock: extObjId = **City-StationHouse.DoorSwitch.Lock**

We see:

a) It's the **purpose of MIDAS Objects to instrument models**

b) **More than one MIDAS Object can be orchestrated to instrument one model**

Now the SMUOS Framework can help in several ways:

1. The SMUOS Framework features a **Model Prototype (MOP "MbBoundModel")**, which helps the author of a bound model to **orchestrate the MIDAS Objects (MOBs) of the model**
2. The SMUOS Framework features **MIBs (MIDAS Bases) for three types of MOBs**, to **ease the implementation of MOBs** by programmers
3. Both (the MOPs and the MIBs) are based on the "MibCore" prototype internally, however this is not visible to the author of the model nor to the programmer of the MOB

All this is depicted in Figure 24, as follows.

The light green parts are actually parts of the SMUOS Framework, the dark green parts are provided by 3rd party programmers and the red part is provided by a 3rd party author.

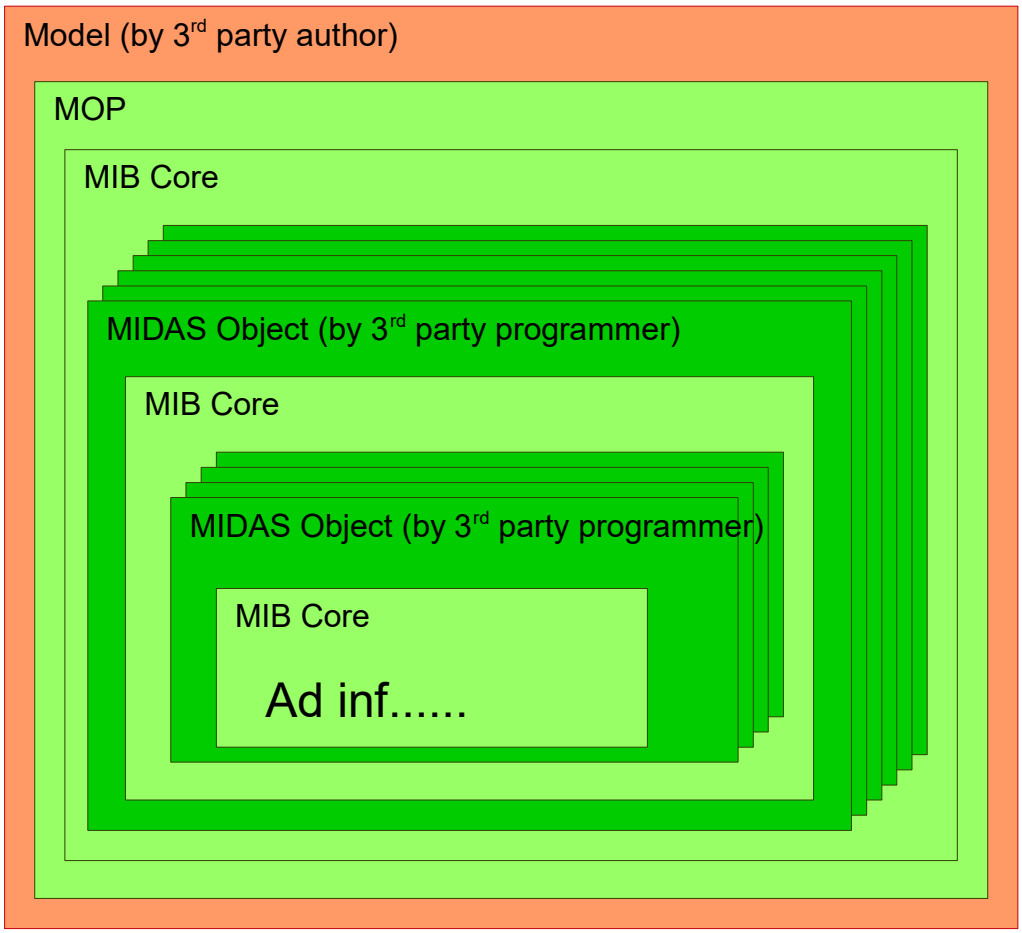


Figure 24: Orchestrating more than one MOB to instrument a model

The orchestration is supported by the MibCore prototype, which cares that all MOBs of a model have got the same mode of operation (MOO) at any time.

Let's have a closer look to the modes of operation (MOOs) in the next chapter.

11.5.2 Modes of Operation (MOOs) of Objects

On the one hand every object (i.e. every model and every MIDAS Object) needs to be attached to a module in order to get access to the information from the Module Activity Matrix (MAM).

Well, not really every object, but some objects may be "astral" objects, which exist outside of all modules.

On the other hand, we are currently planning to implement "unbound" objects in step 0033.11 „Arimathea“ of the SrrTrains v0.01 project. Unbound objects will be able to exist outside of all modules and will be able to change their module (what we will call "handover").

All this led to the definition of four "Modes of Operation (MOOs)" for ongoing operation and one "Mode of Operation (MOO)" for disabled objects.

- MOO I "initialized" astral object
- MOO II "attached" bound object
- MOO III "initialized" unbound object (aka "detached")
- MOO IV "attached" unbound object
- MOO V "disabled"

Immediately after the object has been loaded, it exists in the MOO "LOADED", to be complete.

The possible MOO Changes (changes of MOO) that can be undergone by objects, are explained in more detail in chapter „11.6.4 MOO Changes and Procedures„.

At a first glance, we can settle following statements:

- "Astral" objects are never attached to a module (but they have access to the commParam)
- "Bound" objects are always attached to the their module (they have access to the modParam of that module) and they cannot change the module they are attached to
- "Unbound" objects may be attached to a module or not and they may change the module they are attached to (they may perform a "handover")

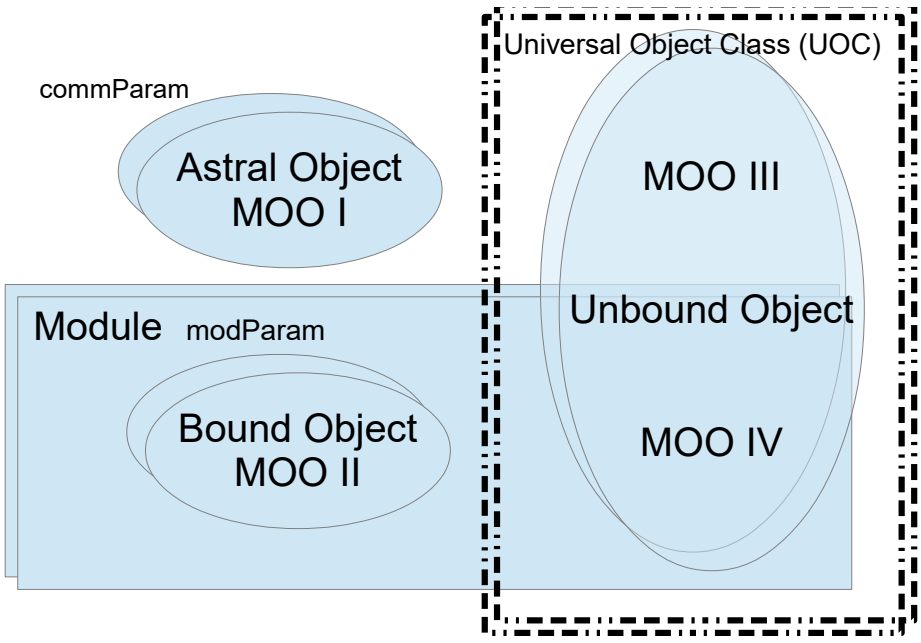


Figure 25: Objects and their modes of operation (MOOs)

11.5.3 Types of MIDAS Objects

The MOO of an object describes, how this object is related to the modules and to the frame of the scene (SMS).

The "type" of a MIDAS Object is a rough classification about how the object handles its global state (this is the set of values that are stored on the central server to ensure they are synchronized among all scene instances, i.e. among all instances of the object).

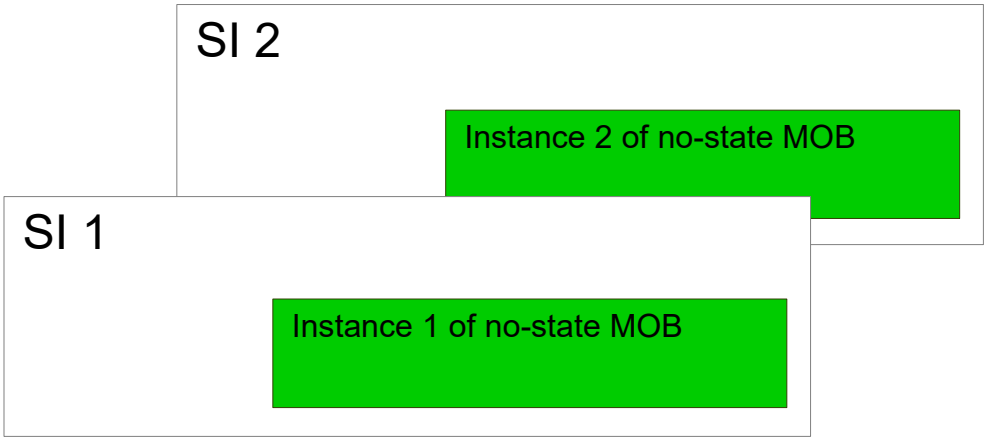


Figure 26: Instances of a no-state MOB don't share a global state

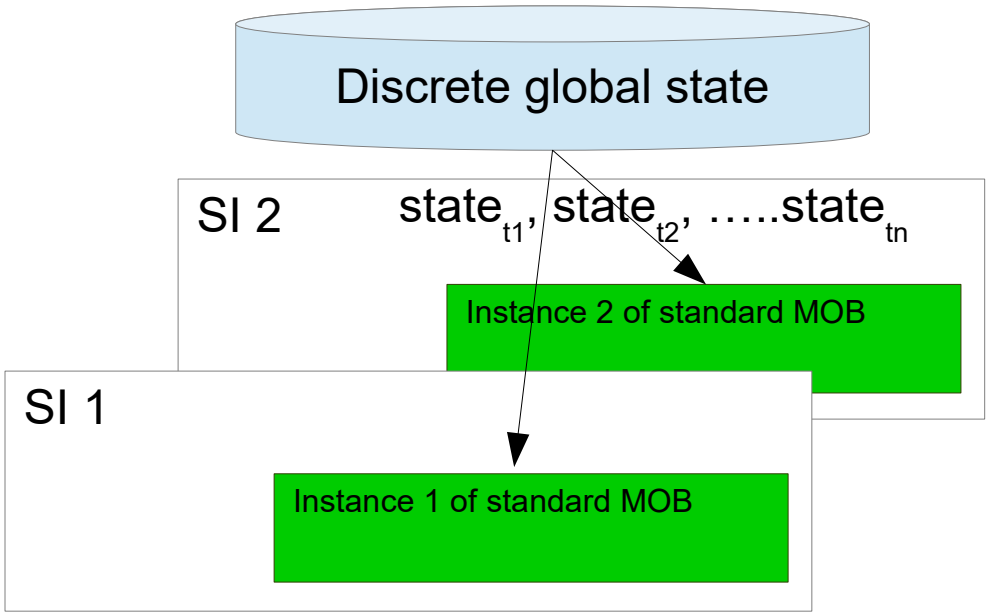


Figure 27: Instances of a standard MOB share a discrete global state

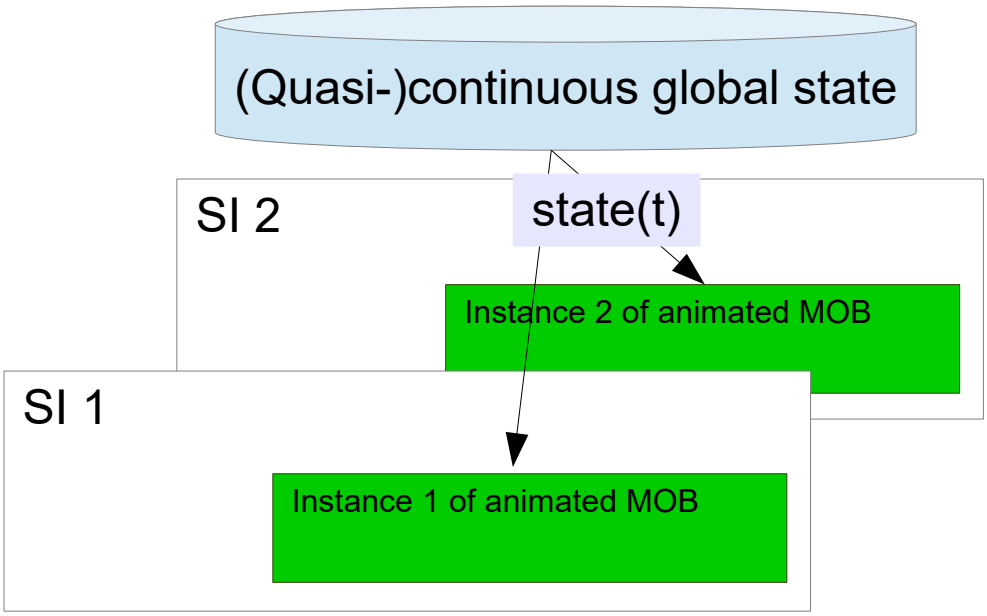


Figure 28: Instances of an animated MOB share a (quasi-)continuous global state

11.5.4 Purpose of the MIDAS Base (MIB)

The purpose of the MIBs is to ease the implementation of MOBs.

We implemented three MIBs, one for each type of MOB:

- MIB for no-state MOBs
- MIB for standard MOBs
- MIB for animated MOBs

Some core functions – which are equal to all MIBs – are implemented in the MIB Core and in the MIB OSM. However, this is of interest only for the developer(s) of the SMUOS Framework.

The next figure depicts all interfaces that you are involved with, when you implement a MIDAS Object (MOB).

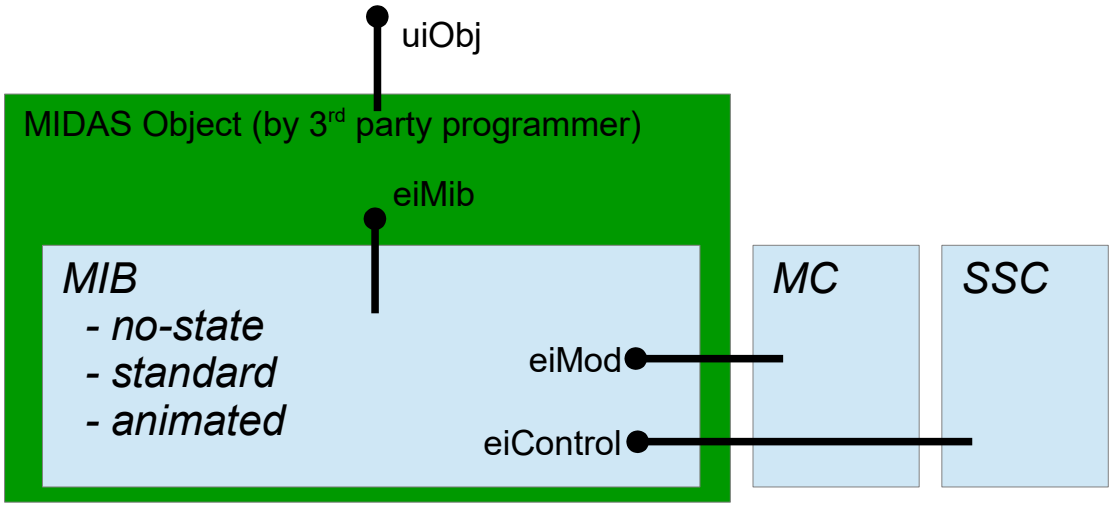


Figure 29: Relevant interfaces for the implementation of a MOB

Each MIDAS Object

- **must** provide the uiObj Interface to its users (model authors, module authors, frame authors)
- **may** use the eiMib Interface of one of the MIBs to ease the implementation of a "no-state", of a "standard" or of an "animated" MOB.
- **may** use the eiMod interface to directly access the MC (either MC Base or MC Extensions)
- **may** use the eiControl interface to directly access the SSC (either SSC Base or SSC Extensions)

11.6 Settled Properties of the Core Prototypes

SscCore

- organizes the **Access to SSC Extensions** by SSC Base and by any SSC Extension
- handles the **MOO Changes** of any SSC Extension
- orchestrates the **MOO Changes** of all dependent SSC Extensions of SSC Base and of any SSC Extension
- ensures the **MOO Changes** of all UOC Related SSC Dispatchers of SSC Base and of any SSC Extension (this includes the initialization of dispatcher stubs for UOC parameters)
- requests the handling of all **Procedures** from its user (i.e. programmer of the SSC Extension) via a call-back mechanism
- triggers the initialization of all **SSC Related Network Sensors** and of all **UOC Related SSC Dispatchers** of SSC Base and of any SSC Extension
- handles initialization failure of any **SSC Related Network Sensor** and of any **UOC Related SSC Dispatcher** of SSC Base or of any SSC Extension

McCore

- organizes the **Access to MC Extensions** by MC Base and by any MC Extension
- handles the **MOO Changes** of any MC Extension
- orchestrates the **MOO Changes** of all dependent MC Extensions of MC Base and of any MC Extension
- requests the handling of all **Procedures** from its user (i.e. programmer of the MC Extension) via a call-back mechanism
- organizes the access to every **required SSC Extension**, as required by any MC Extension

MibCore

- organizes **Nested Objects**
- handles the **MOO Changes** of any object (i.e. of any model or of any MIDAS Object)
- orchestrates the **MOO Changes** of all dependent objects of any object
- requests the handling of all **Procedures** from its user (i.e. programmer of the MIB) via a call-back mechanism
- organizes the access to every **required SSC Extension**, as required by any object
- organizes the access to every **required MC Extension**, as required by any object
- triggers the initialization of all **Object Related Network Sensors** and of all **MIB Related Network Sensors** of any object
- handles initialization failure of any **Object Related Network Sensor** and of any **MIB Related Network Sensor** of any object

11.6.1 Access to SSC Extensions

The following table shows some information, which is **relevant for SSC Base and for any SSC Extension**. It tries to explain, how SSC Base or any SSC Extension can access other SSC Extensions.

Short and sweet, each part of the SSC can access all of its dependent SSC Extensions and each SSC Extension can access the SSC Base via

- some fields that each SSC Extension **must** provide at the **minimum Interface miControl,**
- some fields that **should** be used at the **external interface eiSscCore** and
- some fields of the **commParamExt** that **will** be handled by the SSC Core prototype

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

Interface miControl	Interface eiSscCore	Common Parameter Extension
instanceIdClient SFString	**instanceIdClient SFString**	
instanceIdServer SFString	**instanceIdServer SFString**	
	tracerInstanceIdClient SFString	
	tracerInstanceIdServer SFString	
traceLevelControl SFInt32	traceLevelControl SFInt32	
traceLevelComm SFInt32	traceLevelComm SFInt32	
mandatorySscExtensions MFNode	**mandatorySscExtensionsIn MFNode**	
optionalSscExtensions MFNode	**optionalSscExtensionsIn MFNode**	
	assertedOptionalSscExtensions MFNode	
	commParamExtIn SFNode	
		commParam SFNode
		sscBaseExt SFNode
		extensions MFNode

11.6.2 Access to MC Extensions

The following table shows some information, which is **relevant for MC Base and for any MC Extension**. It tries to explain, how MC Base or any MC Extension can access other MC Extensions.

Each part of the MC can access all its dependent MC Extensions and each Extension can access the MC Base.

It's recommended to <connect> **bold written fields** directly from miMod to eiMcCore.

Interface miMod	Interface eiMcCore	Module Parameter Extension
instanceIdMod **SFString**	**instanceIdMod** **SFString**	
	setModuleName SFString	
	tracerInstanceId SFString	
	tracerInstanceIdClient SFString	
localTraceLevel SFInt32	traceLevelControl SFInt32	
mandatoryMcExtensions **MFNode**	**mandatoryMcExtensionsIn** **MFNode**	
optionalMcExtensions **MFNode**	**optionalMcExtensionsIn** **MFNode**	
	assertedOptionalMcExtensions MFNode	
	modParamExtIn SFNode	
		modParam SFNode
		smsModCoordExt SFNode
		extensions MFNode

11.6.3 Nested Objects

The following table shows some information, which is **relevant for any MIDAS Object**. It tries to explain, how any MIDAS Object can orchestrate all its dependent MIDAS Objects.

Each MIDAS Object can access all its dependent MIDAS Objects.

It's recommended to <connect> **bold written fields** directly from uiObj to eiMib.

Interface uiObj	Interface eiMib
objId **SFString**	**objId** **SFString**
universalObjectClass **SFNode**	**universalObjectClass** **SFNode**
parentObj **SFNode**	**parentObj** **SFNode**
dependentMobs **MFNode**	**dependentMobs** **MFNode**
	addDependentMobs MFNode
	removeDependentMobs MFNode
	assertedDependentMobs MFNode

11.6.4 MOO Changes and Procedures

The following table shows some information, which is **relevant for any SSC Extension**. It tries to explain, how the MOO Changes and Procedures of any SSC Extension are organized.

Short and sweet, SSC Base or other parents of any SSC Extension use the miControl interface to initialize or disable the SSC Extensions, the SSC Extensions use the eiSscCore interface to ease the implementation, because it is defined:

- some fields that each SSC Extension **must** provide at the **minimum Interface miControl,**
- some fields that **should** be used at the **external interface eiSscCore** and
- some fields of the **commParamExt** that **will** be handled by the SSC Core prototype.

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

Interface miControl	Interface eiSscCore	Common Parameter Extension
basicallyInitialized SFBool		
commParam **SFNode**	**commParamIn** **SFNode**	
initialized SFNode	initialized SFNode	initialized SFBool
disable **SFTime**	**disableIn** **SFTime**	
	disabled SFBool	
enabledOut **SFBool**	**enabledOut** **SFBool**	enabled SFBool
	currentProcedure SFString	
	startProcedure SFBool	
	procedureFinished SFBool	

The following table shows some information, which is **relevant for any MC Extension**. It tries to explain, how the MOO Changes and Procedures of any MC Extension are organized.

Short and sweet, MC Base or any other parent of MC Extensions use the miMod interface to initialize or disable MC Extensions, the MC Extensions use the eiMcCore interface to ease the implementation, because it is defined:

- some fields that each MC Extension **must** provide at the **minimum Interface miMod,**
- some fields that **should** be used at the **external interface eiMcCore** and
- some fields of the **modParamExt** that **will** be handled by the MC Core prototype.

It's recommended to <connect> **bold written fields** directly from miControl to eiSscCore.

Interface miMod	Interface eiMcCore	Module Parameter Extension
basicallyInitialized SFBool		
modParam **SFNode**	**modParamIn** **SFNode**	
initialized SFNode	initialized SFNode	initialized SFBool
startAttachment SFInt32	startAttachment SFInt32	
attachmentFinished SFNode	attachmentFinished SFNode	moduleIx SFInt32
disable **SFTime**	**disableIn** **SFTime**	
	disabled SFBool	
enabledOut **SFBool**	**enabledOut** **SFBool**	enabled SFBool
	currentProcedure SFString	
	startProcedure SFBool	
	procedureFinished SFBool	

The following table shows some information, which is **relevant for any MIDAS Object**. It tries to explain, how the MOO Changes and Procedures of any MIDAS Object are organized.

Short and sweet, the parent uses the uiObj interface to initialize and/or to attach or disable the MIDAS Object, the MIDAS Object uses the eiMib interface to ease the implementation, because it is defined:

- some fields that each MIDAS Object **must** provide at the **user Interface uiObj** and
- some fields that **should** be used at the **external interface eiMib.**

It's recommended to <connect> **bold written fields** directly from uiObj to eiMib.

Interface uiObj	Interface eiMib
commParam **SFNode**	**commParam** **SFNode**
initialized SFNode	initialized SFNode
modParam **SFNode**	**modParam** **SFNode**
attached SFNode	attached SFNode
disable **SFTime**	**disable** **SFTime**
	disabled SFBool
enabledOut **SFBool**	**enabledOut** **SFBool**
	currentProcedure SFString
	startProcedure SFBool
	procedureFinished SFBool

11.6.5 Required SSC Extensions

Each SSC Extension **must** define a "well known ID", which has to be used to access the SSC Extension by any MC Extension or by any MIDAS Object:

Interface miControl	Interface eiSscCore	Common Parameter Extension
		wellKnownId SFString

Each MC Extension that **requires** an SSC Extension can require and access the SSC Extension with the help of the eiMcCore interface:

Interface miMod	Interface eiMcCore	Module Parameter Extension
	requiredSscExtensionsIn MFString	
	assertedSscExtensions MFNode	

Each MIDAS Object that **requires** an SSC Extension can require and access the SSC Extension with the help of the eiMib interface:

Interface uiObj	Interface eiMib
	requiredSscExtensions MFString
	assertedSscExtensions MFNode

11.6.6 Required MC Extensions

Each MC Extension **must** define a "well known ID", which has to be used to access the MC Extension by any MIDAS Object:

Interface miMod	Interface eiMcCore	Module Parameter Extension
		wellKnownId SFString

Each MIDAS Object that **requires** an MC Extension can require and access the SSC Extension with the help of the eiMib interface:

Interface uiObj	**Interface eiMib**
	requiredMcExtensions MFString
	assertedMcExtensions MFNode

11.6.7 SSC Related Network Sensors and UOC Related SSC Dispatchers

Each SSC Extension may use SSC Related Network Sensors and/or UOC Related SSC Dispatchers.

Both should be made known to the SSC Core Prototype via the eiSscCore Interface. The SSC Core prototype will care for the MOO Changes of those nodes:

Interface miControl	Interface eiSscCore	Common Parameter Extension
	networkSensorsIn MFNode	
	sscDispatchersIn MFNode	

11.6.8 Object Related and MIB Related Network Sensors

Each MIB may contain MIB Related Network Sensors, however this is something internal to the SMUOS Framework.

Each MIDAS Object may use Object Related Network Sensors. Those network sensors should be made available to the MIB via the eiMib interface. The MIB Core prototype will care for the MOO changes of those nodes.

Interface uiObj	**Interface eiMib**
	networkSensors MFNode

The same is true for bound models, which may use Object Related Network Sensors, too.

Those network sensors should be made available to the MIB Core prototype via the external interface of the Model Prototype (MOP). The MIB Core prototype will care for the MOO changes of those nodes.

11.7 A Possible Evolution Path for the SMUOS Framework

11.7.1 Rebase to Core Prototypes ("Pieta") – done

Software Part		Features
(F)rame	Simple Scene Controller	• init • Support of single-user/multi-user scenes
	SSC Core	• Support of no-state/stateful SSC Extensions • Support of synchronous controllers
(M)odule	Module Coordinator	• commParam + disable • Support of static/dynamic modules
	MC Core	• Support of attacheable/non-attacheable MC Extensions
(M)odel	Model Base	• modParam + disable • Support of bound models
	MIDAS Base	• commParam + modParam + disable • Support of astral/bound MIDAS objects • Support of no-state/standard/animated MIDAS objects

11.7.2 Unbound Models ("Arimathea") – envisioned for 2019

Software Part		New Features
(M)odel	Model Base	• Support of model containers • Support of unbound models
	MIDAS Base	• Support of unbound MIDAS objects

11.7.3 Handover and Moving Modules – rough ideas

Software Part		New Features
(F)rame	Simple Scene Controller	• disable
(M)odule	Module Coordinator	• Support of moving modules
(M)odel	MIDAS Base	• Support of Handover

12 Moving Modules – the eMMF Paradigm

A.D. 2009 – 2017:

„I want to place a wagon on a wagon. Therefore I need tracks on a wagon“

„Hmmmm. Tracks and the vehicles on the tracks are always rendered relative to a module“

„Then I need modules within models“

„Good idea. This way we could make an enhanced MMF paradigm. Module contains models contain modules contain models contain modules contain models and so on“

„Super! This way we could model the universe: particle within body within country within continent within globe within solar system within galaxy and so on“

Moving modules won't be implemented in step 0033. This will be done later.

13 3D Graphics and the Theory of Relativity

A.D. 2009 – 2017:

„X3D-Earth enables us to position cartesian coordinate systems relative to a curved surface of a globe.“

„A funny idea: why not define a framework to position pseudo-euclidean spaces (implementing the special relativity) within Riemann Spaces (implementing the general relativity)?“

„Sounds interesting“

This topic is ffs.

Appendix A The SMS Tracer

The present chapter will try to explain the SMS Tracer

Tbd.

Appendix B The Console Interface

The present chapter will try to explain the Console Interface

Tbd.

Appendix C Extensibility of the SMUOS Framework

The present chapter will try to explain the interface that is provided by the SMUOS Framework in order to be extended by SMUOS Extensions

Tbd.

Appendix D Message Flows – Best Current Practices

The present chapter will try to explain some concepts by examplifying them

Tbd.

Appendix E Reality – the (N+1)th Scene Instance

Anything is real. There is not anything that does not exist. See also the glossary (Appendix G Glossary).

Inspired by a posting on X3D-public, I decided to think about an additional use case of the SMUOS Framework in February 2011. This thinking has not yet been closed.

Given, a multiuser session consists of N scene instances (N >= 1), what about adding an (N+1)th scene instance that would not be inhabited by users, but that would connect the multiuser session to the reality.

Hence we would have N **virtual** realities – one for each user – and one **real** reality, all of which would be synchronized to each other.

Thus we would have following operational modes for the multiuser session:

- Single user mode (N = 1, SCSI does not exist)
- Multi user mode (N > 1, SCSI does not exist)
- Mixed reality mode (N >= 1, SCSI exists)

The present chapter tries to examplify all of those operational modes.

The overall picture is as follows:

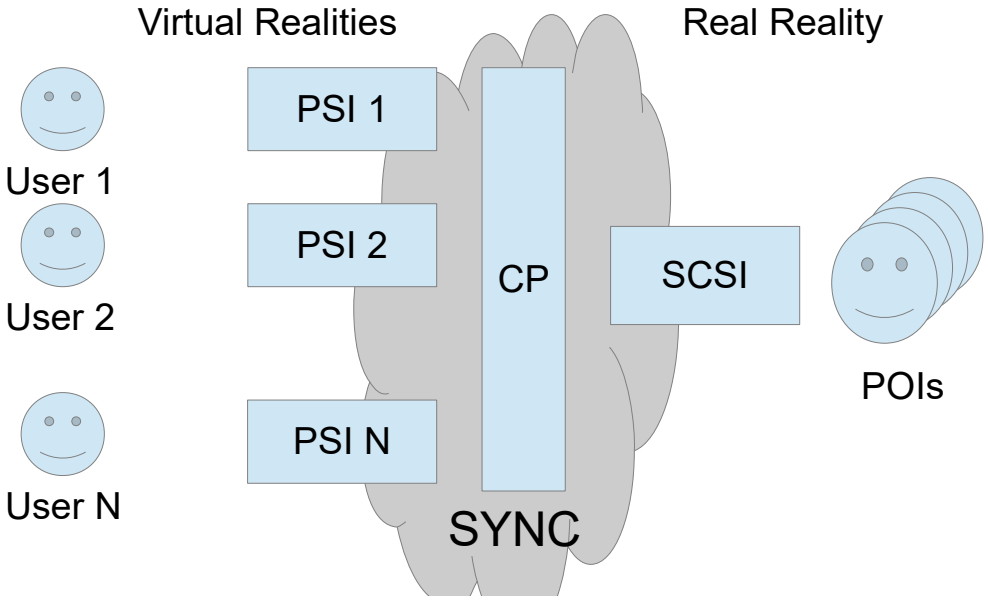


Figure 30: Overall picture of a multiuser session

E.1 Example of a Singleuser Session

An instance of a Simple Multiuser Scene (SMS) can run in single user mode.

Figure 31 depicts all interactions that are possible in single user mode.

Example:

A **user** (e.g. „Alice“)

employs her **senses and skills** (e.g. the abilities to move the mouse and to press buttons on the keyboard or to view the screen) to exchange information with a **personal scene instance** (PSI) – i.e. to inhabit the SMS.

Hence the user attains a **virtual identity** (e.g. „Gandalf“)

and the ability to employ **virtual senses and skills** (vSaSk), e.g.:

- to inhabit the SMS,
- to touch a model,
- to take the ring,
- and so on.

The PSI renders avatars, models and modules, to represent the facilities of the SMS, i.e.

- an **avatar** to represent the virtual identity, e.g. „Gandalfs Avatar“,
  to be exact, we can call it a **virtual life avatar** (VLA),
- **models** to represent the renderable objects of the SMS, e.g. „the ring“,
- **modules** to represent the surroundings of the SMS, e.g. „Mordor“.

Note:

The avatar does not represent the user, but it represents the virtual identity. A small but important difference.

An SMS might enable a user to change the virtual identity without logging out, or it might even enable the user to attain more than one virtual identity in parallel.

Note:

In our example, the mouse, the keyboard, the screen and so on are parts of the personal scene instance (PSI). The interface of the PSI can be used directly by the senses and skills of the user.

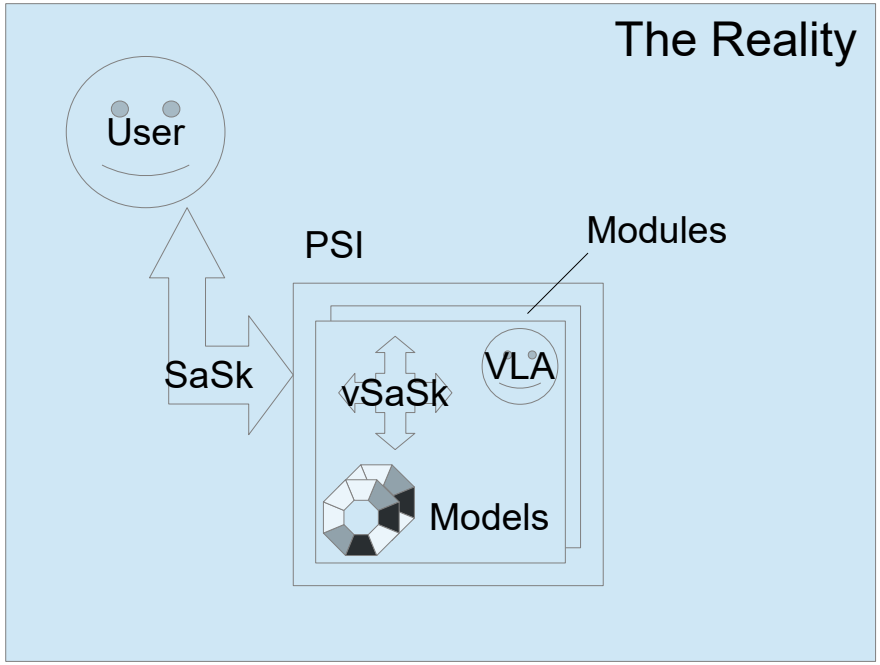


Figure 31: Possible interactions of a multiuser session in single user mode

E.2 Example of a Multiuser Session

A set of instances of an SMS can run in multi user mode.

Figure 32 depicts all interactions that are possible in multi user mode (shown for 2 users).

Running in multi user mode means, every instance of the SMS that runs within the multiuser session, exchanges information with each other instance that runs in the same multiuser session, hence synchronizing the state of all modules, models and avatars and hence keeping the same virtual reality for all N users of the multiuser session.

Example:

> **Synchronizing the State** can mean synchronizing e.g.
>
> - position
> - orientation
> - door open / door closed
> - light on /light off
> - and so on

Note:

> Users may join or leave the multiuser session arbitrarily, hence N is not constant over time.

Note:

> In addition to inhabiting the virtual reality, it may be relevant that all users inhabit the reality, too.
>
> E.g., some of the users could be logged in to a chat room and chat with each other or they could be members of a phone conference to exchange information and/or emotion via voice.
>
> As long as those facilities – we call them **real life facilities** (RLF) – are separate from the facilities of the SMS, we call the multiuser session an **application of virtual reality**.
>
> Otherwise, if real life facilities are synchronized with facilities of the SMS, then we would call the multiuser session an application of **augmented** reality, of augmented **virtuality** or generally spoken of **mixed** reality, see next section.

Note:

> Outside of the virtual reality, **users** represent their **real identities**, within the virtual reality, **avatars** represent the **virtual identities**.
>
> We will see an example in the next section, where a **real life avatar** (RLA) represents a **virtual identity** within the **real reality** – this is really cool stuff :-).

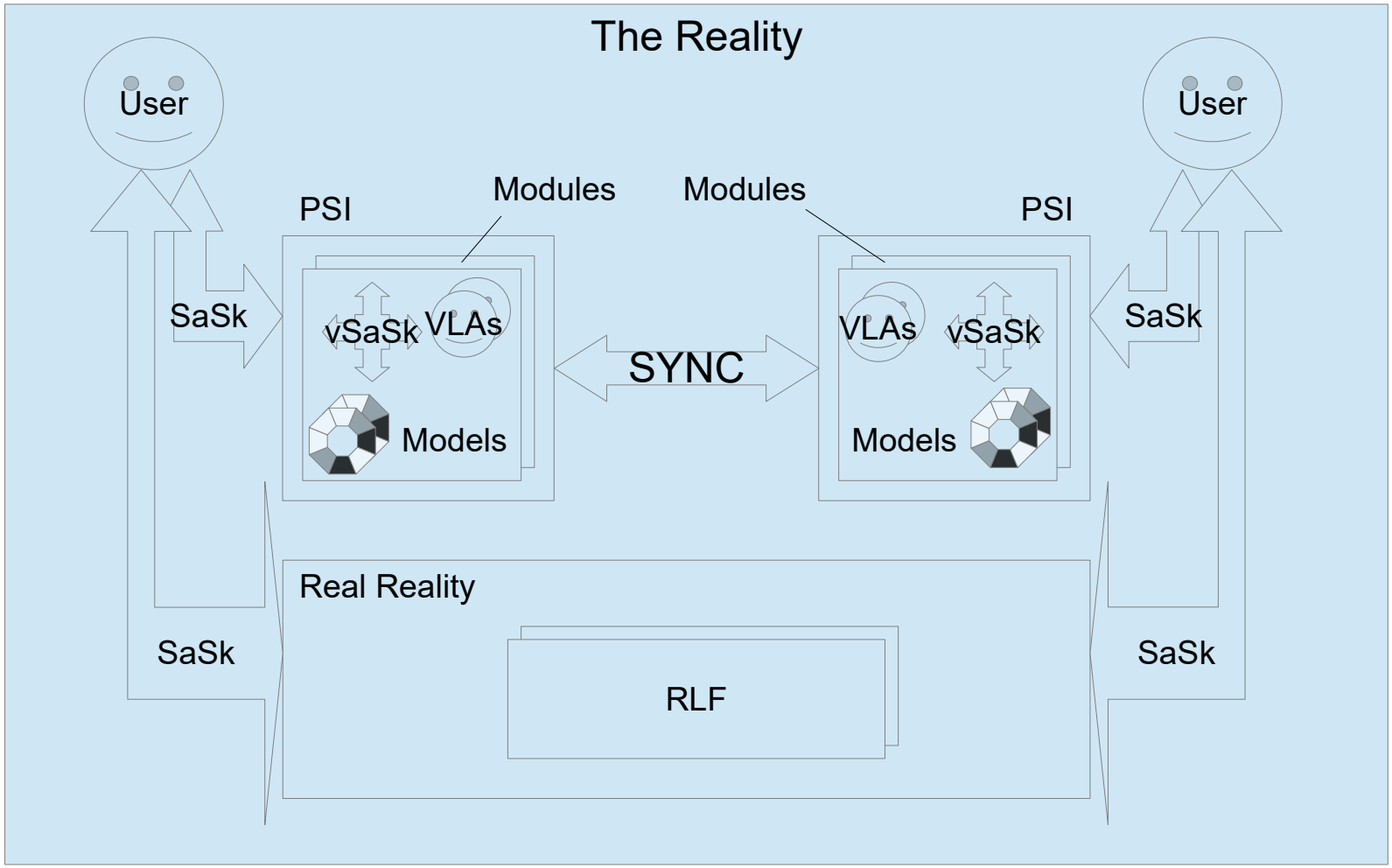


Figure 32: Possible interactions of a multiuser session in multiuser mode (2 users shown)

E.3 Example of a Mixed Reality Session

A set of instances of an SMS can run in mixed reality mode, if at least one facility of the SMS represents a real life facility.

Figure 33 depicts all interactions that are possible in mixed reality mode (shown for 2 users).

Example:

In mixed reality mode we do not use the term real life facility (RLF) – which would be too general – but we distinguish three kinds of RLFs:

- **Real life avatars** (RLA), e.g. a drone,
- **real life objects** (RLO), e.g. airplanes that do not represent virtual identities, and
- **collateral entities** (CE), e.g. a stealth drone of the enemy, as long as not detected.

Now the users attain **remote senses and skills (rSaSk)** that are provided by the RLAs (and optionally by any RLOs) e.g.

- A drone (an RLA) provides the rSaSk of destroying real life facilities
- A remotely controlled car that is modelled in the SMS but that does not represent a virtual identity (i.e. an RLO) provides the rSaSk of crashing into a house

RLFs are **synchronized** with VLFs:

- The state of RLAs is synchronized with their VLAs, e.g. position and orientation
- The state of RLOs is synchronized with their models, e.g. position and orientation
- The state of modules is derived from the universal positioning system (UPS).
  If we assume, e.g., a module being the inner part of a railway wagon, then the wagon spans a local coordinate system that imposes an acceleration to all of its inhabitants (which is equivalent to a gravitational field). Hence the properties of the module are synchronized with the properties of the real wagon (an RLO). The module is a part of the model and the RLO has got UPS properties, it contains a level of the UPS.

Note:

Real life objects (RLOs) and real life avatars (RLAs) are modelled in the SMS and their states are synchronized with the states of the models and virtual life avatars (VLAs).

Collateral entities (CEs) are NOT modelled in the SMS.

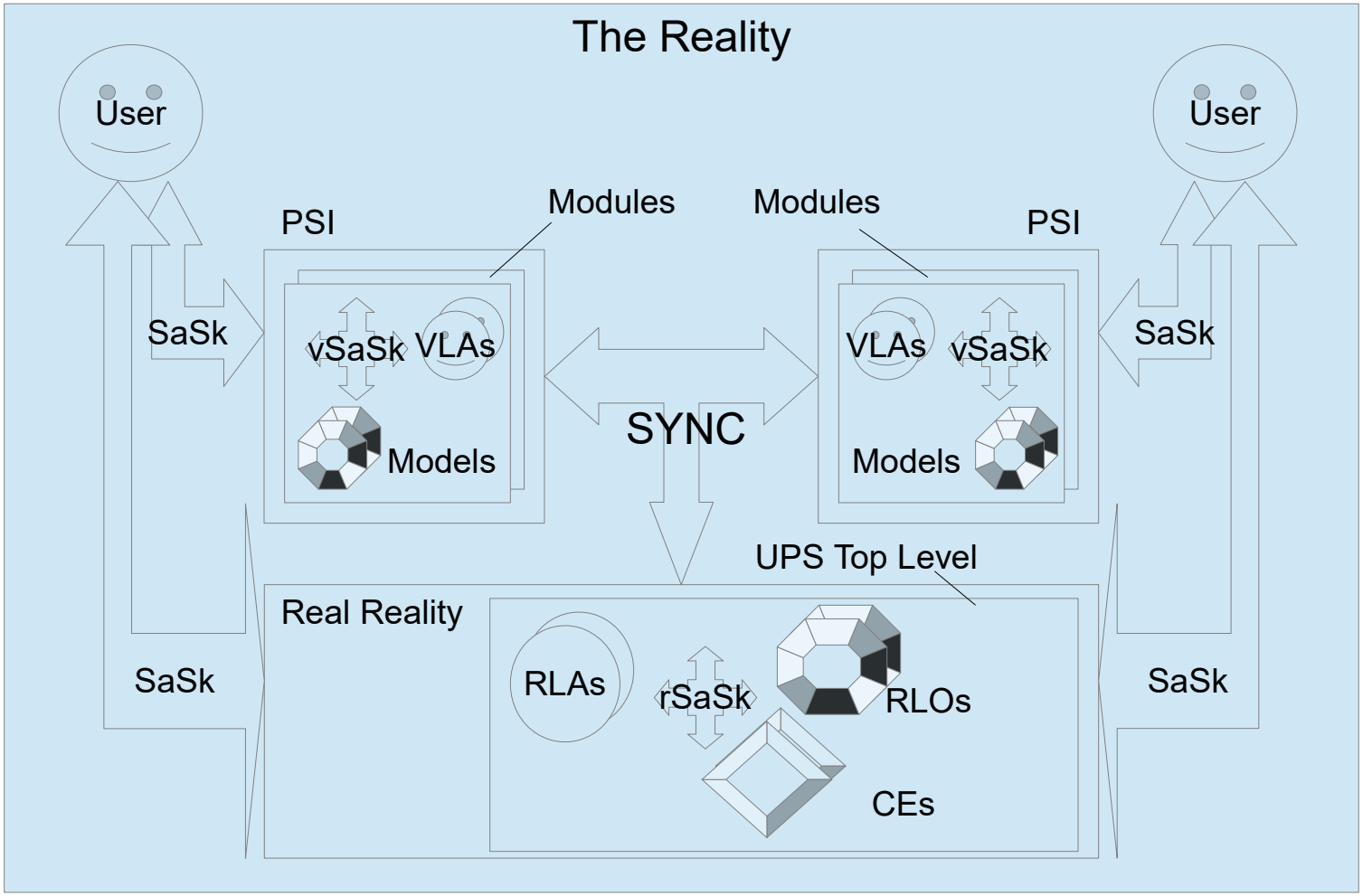


Figure 33: Possible interactions of a multiuser session in mixed reality mode (2 users shown)

E.4 A few Words about Synchronization

In the latest examples we stated the PSIs of a multiuser session are synchronized to each other, if the multiuser session runs in multiuser mode.

In mixed reality mode we would just have an additional scene instance, which would not be a PSI, but which would be the SCSI.

Nice words, sounds simple.

However, synchronization is a little bit tricky, if we want to understand the whole story.

First, we have to understand the **events** and **states:**

An **event** is singular in that it does not depend on history.

Something happens in one scene instance and the scene instance informs all scene instances that this thing has happened and that's it.

Example:

> User „Alice“ touches a model and the model starts an animation in all scene instances (that of „Alice“ and that of „Bob“).

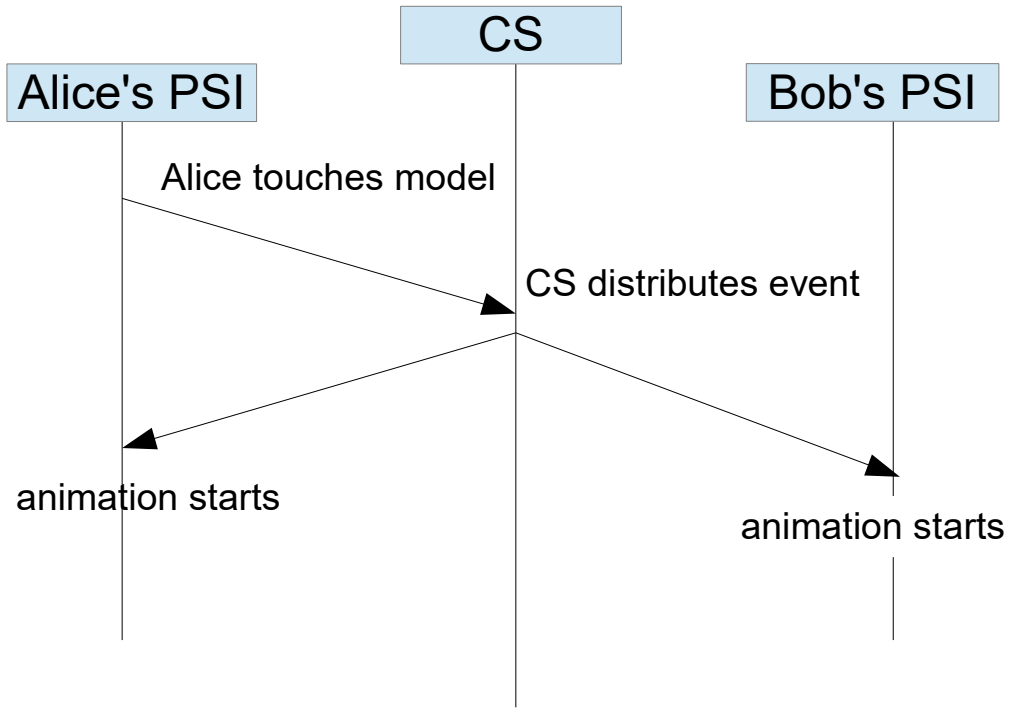


Figure 34: Example of the distribution of an event

A **state** is stored persistently on the collaboration server (CS).

Changes of a state are relative to the current value of the state.

Changes are implemented by **server side calculations**.

Example:

„Alice“ resets the position of a car.

Then „Bob“ joins the session, gets the current value of the state during initialization and afterwards changes the position of the car again.

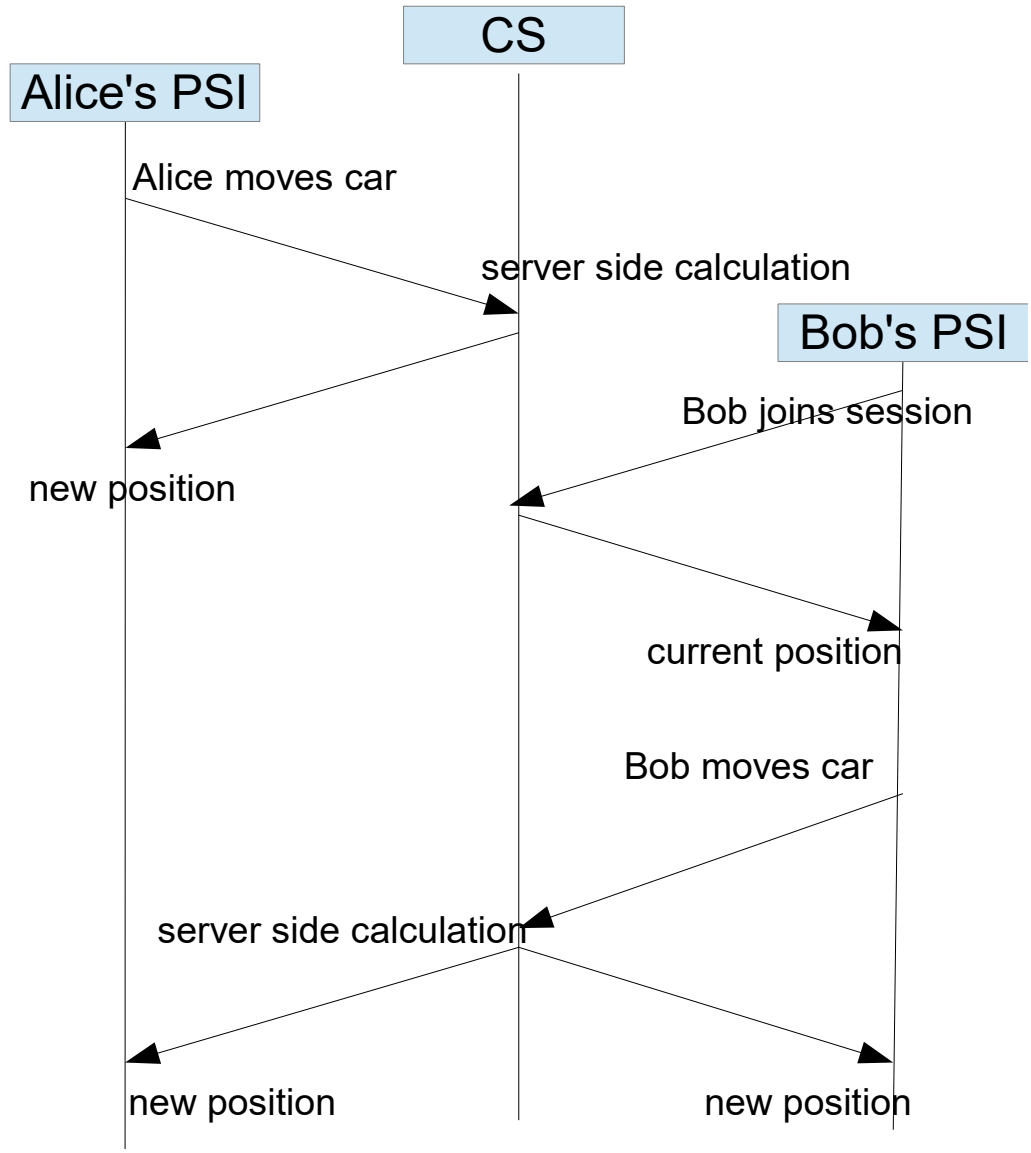


Figure 35: Example of changing a state with server side calculations

Second, we have to understand **MIDAS Objects:**

MIDAS Objects add one level of indirection to the principles of states and events.

That means, the calculations are not performed as server side calculations but they are performed as client side calculations.

The CS is only used to cache the global state of a MIDAS Object. The state is actually stored in the scene instances in the instances of the MIDAS Object.

Only one scene instance is responsible to calculate the global state of a MIDAS Object. We say this instance has got the **Object Controller Role** (OBCO Role), or shortly **it is the OBCO for this MIDAS Object.**

Example:

„Alice“ changes the position of the car. „Bob“'s PSI is the OBCO of the car.

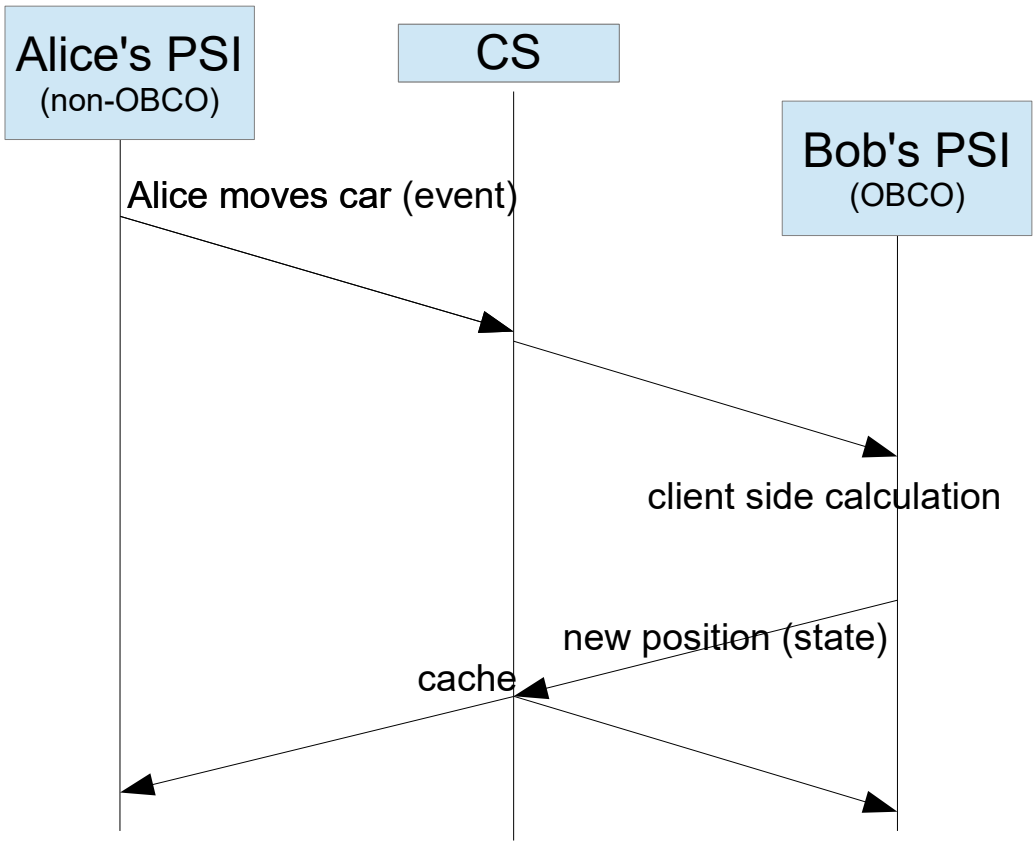


Figure 36: Example of changing a state with client side calculations

We can depict the principle of MIDAS Objects in a layered diagram

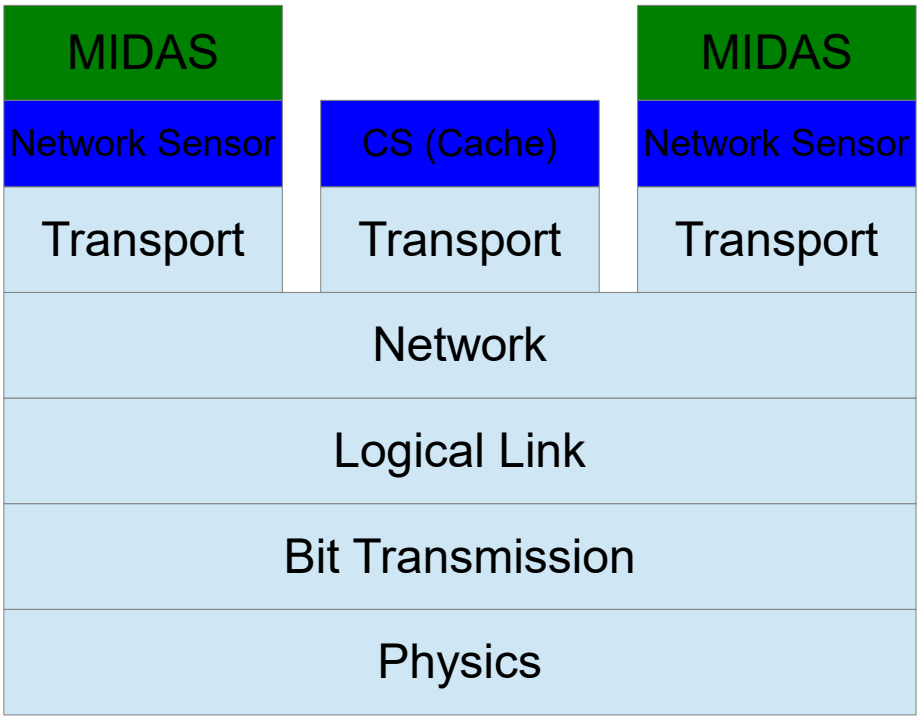


Figure 37: Layered view of MIDAS object and CS within multiuser session

Third, we have to understand **POIs**

POIs are the real life counterparts of MIDAS Objects. MIDAS Objects reside in the 3D Web, POIs reside in the IoT. This is ffs.

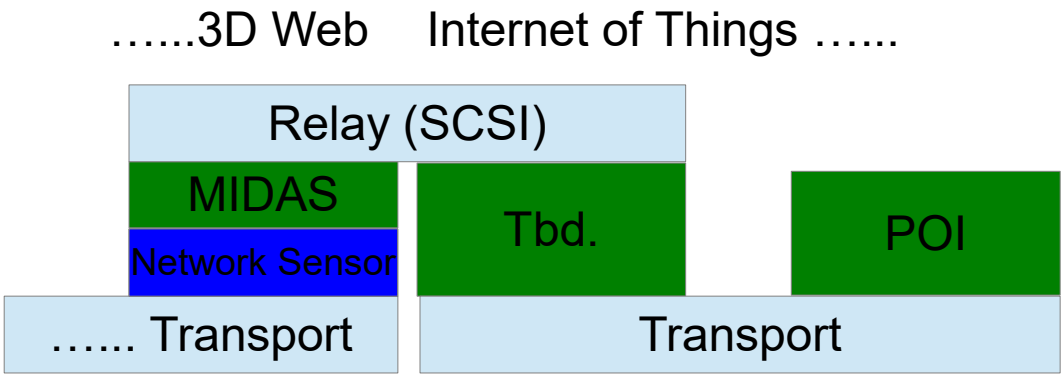


Figure 38: The relation between MIDAS Objects and POIs

Appendix F Trying a Prophecy about SrrTrains v0.01

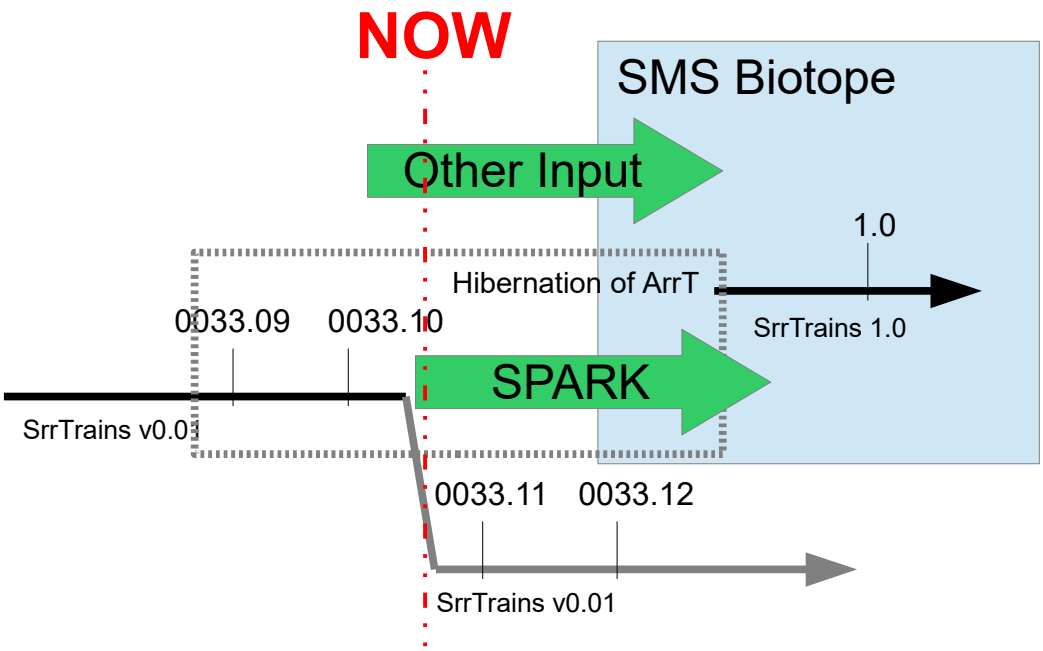
SrrTrains will die – and it will rise again

Not so metaphoric:

The results of SrrTrains v0.01 will be taken as input

to create a basis (what-I-call „SMS Biotope“),

which can be used for many applications (and for SrrTrains v1.0)



- SrrTrains v0.01.....http://letztersein.wordpress.com/srrtrains-v0-01
- SPARK.....https://github.com/christoph-v/spark
- SrrTrains v1.0.....to be defined

Appendix G Glossary

Reality, Virtual Reality, Real Reality

There is only one **reality**, but every person carries an own **model of the reality** in his mind. This model helps the person to foresee the future development of reality and it helps the person to influence reality according to his will.

Virtual reality is a part of the reality that is implemented by means of technology and that helps one or more persons (see **user**) to inhabit a virtual scene that needs not be directly related to the reality.

Strictly spoken, an ancient form of virtual reality is already to sit around the camp fire telling stories. Also books and movies form kinds of virtual reality.

Usually we use the **narrow term virtual reality**, if some minimum technological requirements are fulfilled, e.g. the usage of stereoscopic computer graphics.

We use the term **real reality** to denote all parts of the reality that are not part of the virtual reality in question, but that are of relevance for the virtual reality.

Anything else is just reality.

User

A **user** is a person who uses a personal scene instance (see below) to inhabit a Simple Multiuser Scene (see below) in the course of a multiuser session (see below).

Personal Scene Instance (PSI)

A personal scene instance is the collection of all technological facilities that are needed so that one user can inhabit a Simple Multiuser Scene.

One important facility of the PSI can be a Web3D browser that interprets a concrete scene graph.

The user interface of the PSI can be used via the senses and skills (SaSk) of the user.

Simple Multiuser Scene (SMS)

A Simple Multiuser Scene is a collection of facilities that are accessible via the 3D Web and that can be instantiated within PSIs to provide virtual senses and skills (vSaSk) to users.

Such facilities include, e.g. (see below for detailed definitions):

- Avatars to be able to represent virtual identies
- Models to be able to render the renderable objects of the scene
- Modules to be able to render the surroundings of the scene
- Geographic infrastructure to be able to render the surroundings of the scene

Multiuser Session

A multiuser session is an instantiation of an SMS for a concrete set of users.

Those users will be able to inhabit the virtual scene together.

Technically spoken, a multiuser session is a collection of one or more PSIs and of one optional SCSI (see below), all of which are synchronized to each other.

Server/Controller Scene Instance (SCSI)

The Server/Controller Scene Instance connects the multiuser session to the real reality in order to synchronize real life facilities (see below) with virtual life facilities (see below).

This enables the mixed reality mode to be used as operational mode (see below).

Virtual Life Facility (VLF)

Virtual life facilities are used to provide virtual senses and skills to a user. In mixed reality mode VLFs may be synchronized to real life facilities (see below).

A VLF is an instantiation of a facility of the SMS.

Examples of VLFs are:

- Virtual life avatars (or simply avatars) to represent virtual identities to one user
- Models to render the renderable objects of the scene to one user
- Modules to render the surroundings of the scene to one user
- Geographic infrastructure to render the surroundings of the scene to one user

Real Life Facility (RLF)

Real life facilities are parts of the real reality.

We distinguish following kinds of RLFs: real life avatars (see below), real life objects (see below) and collateral entities (see below).

Operational Modes (OM)

A multiuser session can operate in one of following modes:

- Single user mode – only one PSI exists, SCSI does not exist
- Multi user mode – more than one PSI exist, SCSI does not exist
- Mixed reality mode – at least one PSI exists, SCSI exists

Model, Real Life Object (RLO)

A model is an object within an SMS that can be rendered.
In other words, it is an object to the virtual senses and skills of the user, when he inhabits the SMS through the PSI.

In mixed reality mode, a model may represent a real life object (RLO).

An RLO is always represented by a model, otherwise it would be a collateral entity.

Avatar, Virtual Life Avatar (VLA), Real Life Avatar (RLA)

An avatar is an object that represents a virtual identity (see below). A virtual life avatar is a model that represents a virtual identity and a real life avatar is an RLO that represents a virtual identity.

Collateral Entity (CE)

A collateral entity is an RLF that is not an RLO. I.e. it is a real life facility that is somehow relevant for the multiuser session, but it is not modelled in the SMS.

Module, Universal Positioning System (UPS)

According to the MMF paradigm, an SMS consists of one or more modules that build the surroundings of the scene, whereas each renderable object (each model) is assigned to one of the modules.

A module spans a local (pseudo-) euclidean spacetime, which is used to position the models.

In mixed reality mode, we will often use WGS84 coordinates as global coordinates, which can be used to position the modules.

Hence a local coordinate system in real reality can be defined relative to GPS coordinates.

Now the SMUOS framework aims to be a framework for the 21st century and hence a GPS will not be enough. We will need something that includes the universe into its concepts, not only the globe.

UPS the right wording for such idea.

And it need to be hierarchical, according to the eMMF paradigm. One level of modules being the top level (within a scene) containing top level models. Each top level model may contain second level modules containing second level models and so on.

Clear, there is nothing like a „top“ level in universe (in UPS), Hence the top level must be identified by gravitational field instead of velocity and position. This is ffs.

Geographic Infrastructure, Tiles

The relations among modules, geographic infrastructure and tiles are ffs.

Identity, Virtual Identity, Real Identity

Need not be defined. If we need to explain this, then we do really have a problem.

Synchronization

SrrTrains uses the Network Sensor / Event Stream Sensor for synchronization of scene instances.

The objects that are used within PSIs and within the SCSI to synchronize the multiuser session, are called MIDAS Objects (Multiuser Interactivity Driven Animation and Simulation Objects).

The SCSI can be seen as relay between MIDAS Objects and the Internet of Things (IoT), where we define POIs (see below) as the peers of the SCSI, when it relays the IoT into the multiuser session.

Point of Interaction, Point of Interest (POI)

A POI is a unit that can be addressed within the IoT.

A Point of Interest delivers a stream of events to the multiuser session. This stream describes (a part of) the state of one or more RLOs.

A Point of Interaction accepts a stream of events from the multiuser session. This stream influences (a part of) the state of one or more RLOs.

A Point of Interaction may deliver a stream of events to the multiuser session. This stream describes (a part of) the state of one or more RLOs.

Connectivity Platform (CP)

X3D scenes communicate with Collaboration Servers (CS) through Network Sensors / Event Stream Sensors. CP is a conceptual name for an evolved CS.