



MOSIM

End-to-end Digital Integration based on Modular
Simulation of Natural Human Motions
ITEA 3, 17028



Project Coordinator: Thomas Bär, Daimler AG



End-to-end Digital Integration based on Modular Simulation of Natural Human Motions

ITEA 3 – 17028



Work package 2

Technical Requirements and Concept for Modular Simulation



Deliverable 2.2

MMU Concept and Interface Specification

Document type	: Deliverable
Document version	: 1
Document Preparation Date	: 2019-09-30
Classification	: public
Contract Start Date	: 2018-11-01
Contract End Date	: 2021-08-31
Author	: Felix Gaisbauer

	<p style="text-align: center;">MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028</p> <p>Project Coordinator: Thomas Bär, Daimler AG</p>	
---	---	---

Final approval	Name	Partner
Review Task Level	Felix Gaisbauer	Daimler AG
Review WP Level	Christian König	TWT
Review Board Level	Klaus Fischer	DFKI

	<p>MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028</p> <p>Project Coordinator: Thomas Bär, Daimler AG</p>	
---	---	---

Executive Summary

This public document is the initial version of deliverable 2.2 “MMU concept and interface specification” from work package 2. The aim of this public document is:

- To introduce the overall concept of the MOSIM approach and its architecture.
- To present a detailed overview of the proposed interfaces and formats.
- To outline the workflow and utilization of the novel framework.
- To present a first proposal for the planned standard, which serves as base for the future MOSIM activities and which can be used to implement the overall framework.

Note that, the **document** is considered as a **living and dynamic document** which is updated throughout the progress of the MOSIM project. In particular, the interfaces, formats and workflows, as described within this document are a first draft and are utilized as a starting point for the implementation phase of the MOSIM project. However, feedback and gained knowledge during the implementation will be fed back to this document, whereas the proposed framework will be adjusted. Overall, a new version of the document will be provided at least once a year.

The overall document is structured as follows:



First in Section 1, an introduction regarding the motivation, basic concept and benefits of the MOSIM approach, as well as the Functional Mock-up Interface (FMI) standard is given. Followed in Section 2, a detailed overview of the overall MOSIM concept, subdivided into MMUs, co-simulation and behavior modeling is given. In Section 3, the technical architecture is revisited in detail. In this context, first the overall framework and the contained components are introduced. Next, a detailed overview of the formats and interfaces is given. Moreover, the workflow of the proposed components is presented in detail. Section 4 furthermore gives a summary and conclusion of the overall document. Additionally, in the Appendix example files, which help in understanding the overall framework, are provided. To be able to implement the proposed framework, the Apache Thrift interface definition files, as well as the launcher application is also provided in addition to the document.

Contents

Executive Summary	3
Contents	4
1 Introduction.....	6
2 Overall concept	7
2.1 Motion Model Units	7
2.2 Co-Simulation	8
2.3 Behavior Modeling & Execution	9
3 Technical Architecture.....	12
3.1 Overview.....	12
3.1.1 Motion Model Unit.....	12
3.1.2 Adapter	13
3.1.3 Target Engine.....	13
3.1.4 Communication Layer.....	13
3.1.5 Middleware	13
3.1.6 Services.....	14
3.1.7 Launcher	14
3.1.8 Intermediate Skeleton.....	14
3.2 Interfaces & Formats.....	15
3.2.1 Motion Model Unit.....	15
3.2.2 Co-Simulation	19
3.2.3 Behavior Modeling & Execution	21
3.2.4 Core Formats	32
3.2.4.1 Motion Instruction (MInstruction)	32
3.2.4.2 Motion Instruction Result (MInstructionResult)	34
3.2.4.3 Constraints (MConstraint)	34
3.2.4.4 Simulation State (MSimulationState).....	39
3.2.4.5 Simulation Result (MSimulationResult).....	40
3.2.4.6 Simulation Event (MSimulationEvent)	41
3.2.4.7 Scene Manipulations (MSceneManipulation)	42
3.2.4.8 Further formats	44
3.2.5 Avatar Representation (MAvatarPosture, MAvatarPostureValues)	48
3.2.6 Scene	54



3.2.7	Adapter	67
3.2.8	Services	69
3.2.9	Launcher (MMIRegisterService)	74
3.3	Workflow	77
3.3.1	Launcher – The entry point for the overall MMI framework	77
3.3.2	MMU Execution	78
3.3.3	Adapters	80
3.3.4	Co-Simulation	83
3.3.5	Behavior Execution	84
3.3.6	Target Engine	86
3.3.7	Skeleton Utilization	87
4	Summary and conclusions	89
5	References	90
6	Appendix	92
6.1	Exemplary MMU Description File	92
6.2	Exemplary file of the Skeleton	93

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

1 Introduction

The overall goal of the MOSIM project is to realistically simulate complex human motions in the context of different use-cases. In academia and on the market, up to now, only isolated digital human simulation approaches are available. For instance, individual tools can already address the simulation of setup paths or ergonomic validation. However a comprehensive simulation of manual assembly scenarios comprising heterogeneous activities is not possible yet. Nonetheless, given the available tools and motion synthesis approaches, most of the requested simulation capabilities are already available. A major hurdle for combining the available technologies is the lacking availability of source code and expertise and uneconomically high porting effort. To allow the open and efficient utilization of these technologies and tools for a comprehensive simulation and benchmarking, the MOSIM projects targets to provide an open standard for connecting heterogeneous digital human simulation approaches in a common framework. In particular, the efforts for incorporation and implementation should be minimized, whereas the major programming languages and platforms shall be supported. Ultimately, the MOSIM framework enables the end-user to combine the best motion synthesis approaches available. Given the MOSIM framework, a new value chain comprising different roles is generated. In particular, vendors of simulation software can sell comprehensive simulation environments. Moreover, a new business for selling digital human simulation approaches in a modular way is created.

For exchanging simulation functionality in a different domain than motions, a widely used solution named Functional Mock-up Interface (FMI) is already available. The proposed MOSIM framework is strongly inspired by the FMI approach. The Functional Mock-up Interface is a standard that supports the exchange of dynamic simulation models as well as its co-simulation while being tool independent. This standard is based on a combination of xml-files and compiled C-code [1]. An instance of an FMI component is called a Functional Mock-up Unit (FMU). By using the FMI standard, it is possible to perform a simulation of different FMUs, containing appropriate solvers, whereas only the simulation results of the FMUs are exchanged after defined time steps. This approach is called FMI for co-simulation [2]. The concept of modular motion units as derived in the MOSIM project, which is also referred as Motion Model Interface (MMI) approach, builds upon the idea of the FMI concept to further extend the standard to simulate human motion.

Orchestrating various sub-simulations as intended by the FMI or MMI approach requires a superior instance managing the distributed sub-systems. In general, this orchestration process is named co-simulation, whereas the co-simulator updates the components and incorporates the results. Recently, in literature various co-simulation approaches for the FMI standard have been proposed ([3] [4] [5]), however, these systems predominantly focus on signal flow modeling mainly in the mechatronic domain. Since the co-simulation of digital human simulation systems has entirely different requirements, these solutions cannot be directly used.

To implement the aforementioned aspects, a standardized framework satisfying the heterogeneous requirements of the digital human simulation approaches, co-simulation, behavior modeling and use-cases is required. The document is structured as follows: Section 2 proposes the overall concept of the MOSIM framework, depicting the main concepts and components. In section 3, the proposed technical architecture is presented in detail. In particular, in section 3.1 the main technical framework and the components are revisited. Section 3.2 focuses and the description of the respective formats and data structures by means of class diagrams. Section 3.3 explains the overall process and workflow of the framework. Section 4 gives a conclusion and summary of the presented framework. Moreover, in the appendix additional example files are provided.

2 Overall concept

Strongly inspired by the FMI approach, a new concept for combining heterogeneous digital human simulation approaches is developed in the MOSIM project. Based on the FMI approach, a concept for exchanging character animation systems is introduced in [6], [7]. With FMI, complex systems like industrial machines can be simulated using specialized approaches such as solvers for pneumatic cylinders or kinematic models. The respective sub-simulations are embedded within standardized modules (FMUs) [2]. A co-simulator sequences several of these co-simulations. This component communicates with the FMUs at discrete points in time and incorporates the computed results of all heterogeneous approaches in a common simulation. Transferring this concept to the domain of character animation, so called Motion Model Interfaces (MMIs) and their implementations called Motion Model Units (MMUs) are presented which allow incorporating diverse character animation approaches into a common framework. Figure 1 shows the main idea of the novel MOSIM approach. In the following, the overall framework is also referred as MMI framework.

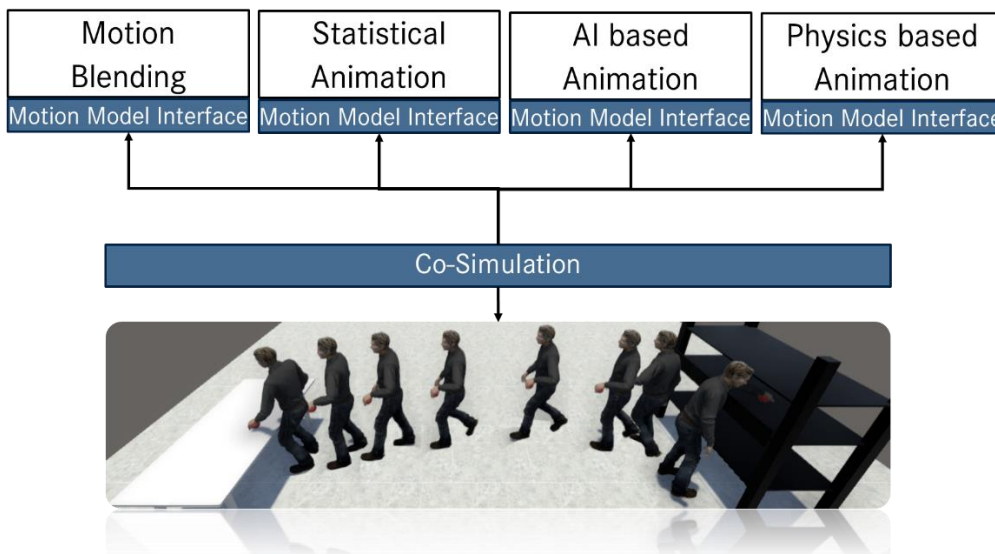


Figure 1 Schematic illustration of the MMI Concept. By encapsulating different digital human simulation approaches in modular units, comprehensive simulations can be realized.

2.1 Motion Model Units

The proposed MMUs are an essential part of this modular concept and provide the basic interface for encapsulating different character animation systems and technologies (see Figure 1 top). These units contain the actual animation approach, being implemented in the required platform and programming language. For instance, an actual MMU can comprise a data-driven algorithm implemented in Python, as well as model-based approaches realized in C++. By utilizing a common interface, and inter-process communication, the MMUs can be accessed independent of the language of implementation. Thus, the communication and workflow are only driven by the functionality provided by the interface and not by specific environments. Figure 2 gives an overview of the provided key functionality of the interface.

Individual MMUs are responsible for generating specific kinds of motion (e.g. locomotion behavior or grasp modeling). Each MMU provides the functionality to set the intended motion instruction, as well as evaluating prerequisites and getting boundary constraints for executing the motion. Moreover, the MMUs comprise a *DoStep* routine that is executed for each frame to be simulated. In this context, the actual posture at the given frame is computed by the specific technology.

For each frame, the MMU provides output parameters describing the generated posture, its constraints, as well as intended scene manipulations and events. Since most motion generation approaches strongly rely on spatial information of the environment and the digital human representation, the communication with the scene is an important aspect for realizing such an encapsulation. Thus, each MMU can access the information provided by the scene through a defined interface (see Figure 2 scene access). In this way, the actual scene representation can be embedded in diverse target environments. Considering the concurrency between different MMUs, manipulations of the scene, which are intended by the MMUs, are not directly written back to the scene; instead, these are provided as an output of the simulation step and are furthermore processed by a superior instance.

Even though the MMU implementations of different motion types such as grasping or walking might be different in terms of the underlying technique and algorithms, frequently, the MMUs utilize similar functionalities such as computing inverse kinematics or planning a path. For this reason, each MMU can additionally access a set of predefined services such as inverse kinematics, retargeting and path planning (see Figure 2 service access).

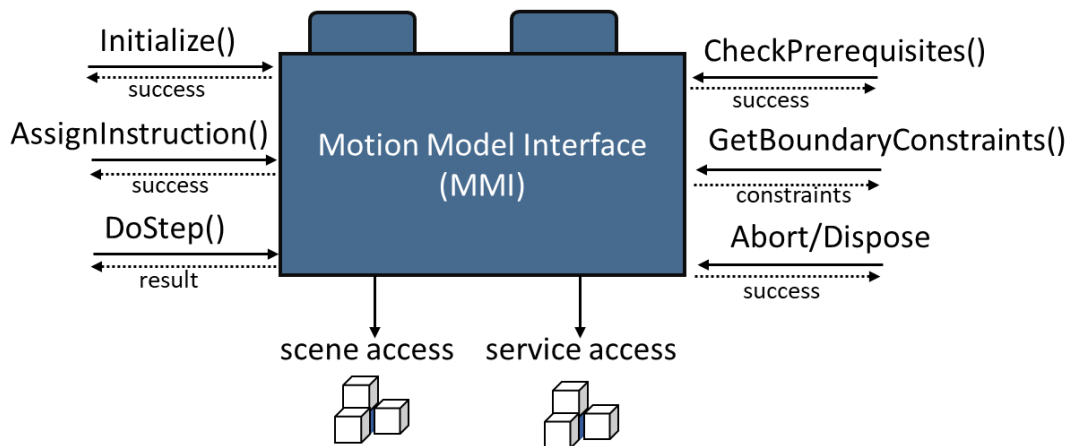


Figure 2 Overview of the basic interface of a so-called Motion Model Unit.

2.2 Co-Simulation

Having distinct MMUs comprising specific simulation approaches, separately generated postures must be merged and further processed to obtain natural motions. Therefore, a co-simulator is required, which orchestrates the actual execution of the MMUs. In this context, the component merges and overlaps the motions, while considering the constraints of the postures. Figure 3 provides an overview of the general input and output of a co-simulation. Generally, the input of the co-simulation is a list of different tasks (e.g., walk to, grasp object) with temporal dependencies. The output of the co-simulation is a feasible motion representing the specified tasks.

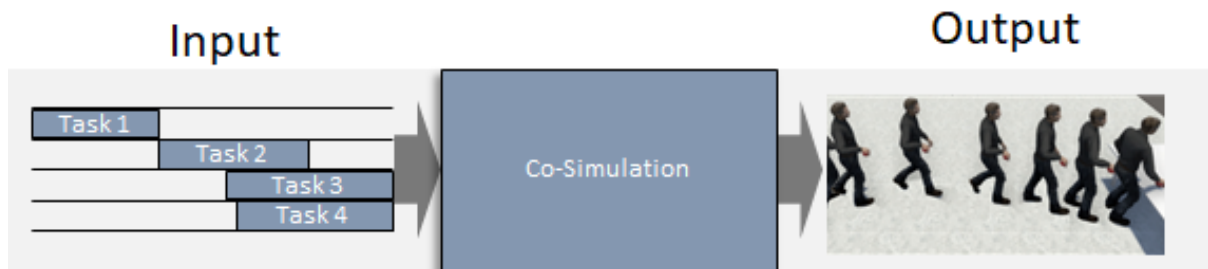


Figure 3 Schematic illustration of the input and output of a co-simulation as required for the MOSIM framework.

Since the scope of the framework is to incorporate extensively heterogeneous character animation systems, the individual MMUs might comprise entirely different skeleton structures and anthropometries. To utilize these heterogeneous results in a common platform, retargeting to a global reference skeleton is required for each MMU. Moreover, since two consecutive MMUs might start/end with a different posture (e.g., MMU1 ends with t-Pose, MMU2 starts with idle pose), the transition between the respective units must be explicitly modeled. Another essential aspect of the co-simulation is to preserve the constraints and characteristics of the original motions/ postures (e.g., Grasp MMU1 requires the hand to be at a specific location).

One possibility to handle different MMUs is to sequentially execute them. However, when examining humanoid motion, it can be encountered that most of the performed motions are commonly executed in parallel. Therefore, an important task of the co-simulation is to overlap and merge different motions generated by the MMUs.

In [8], a first concept of a co-simulation for digital human simulation was presented. Within the MOSIM project, different concepts and variations of possible co-simulation approaches will be investigated. The co-simulation interface proposed in this document forms the basis for the future implementations and investigations.

2.3 Behavior Modeling & Execution

In order for the co-simulator to be able to merge different motions, in the first place, it requires a sequence of instructions or MMUs to be executed as input. Which motions or granular tasks an avatar has to execute depends on two factors: on the one hand on the task definition to be simulated and on the other hand on the current state of the avatar and/or the simulated environment. Depending on these factors, there must be a mechanism that considers both together and then reasons which granular task to execute. In the MOSIM project, the Behavior Modeling & Execution unit is responsible for this reasoning task. It acts on a semantic level at which higher-level actions or behaviors are executed and thus serves as a mediator between the co-simulator and the holistic scenario to be simulated.

Task descriptions defined purely as sequences have the restriction that alternative activities cannot be represented. In a dynamic environment in which more than one entity (e.g., avatar, user, etc.) interacts with the simulated environment, there are different options to perform higher-level actions, which requires an approach that allows mapping of alternative context-dependent activities.

In order for an entity to be able to make decisions independently and to be able to carry out context-dependent actions, in addition to a behavior model knowledge representation is required to determine the current state of the entity. Additionally, constraints for the selection or execution of given MMUs or actions can guide the reasoning in the Behavior Modeling & Execution unit. After a context-dependent action has been executed, feedback on the result in the environment is needed. This feedback is received either via an update of the scene state or from the co-simulator in case errors (e.g., an MMU could not make an avatar reach the desired position) in MMU execution are reported.

Furthermore, it must be easy for a developer to model a behavior or to adapt it to a considered task description to be able to run through different scenarios as quickly and intuitively as possible.

A popular approach to achieve this flexibility is given by the Behavior Tree (BT) paradigm, which is a graphical programming language combining Decision Trees with Finite State Machines and has its roots in the gaming industry. BTs are also used extensively in robotics to realize autonomous and goal-driven entities that decide reactively which actions are to be performed depending on their current situation. Figure 4 shows for example a Reasoning Cycle for a simulated worker controlled by a BT. In the first step, the current situation of the worker is perceived and then evaluated with a BT. The next action is derived and executed, resulting in a new situation.

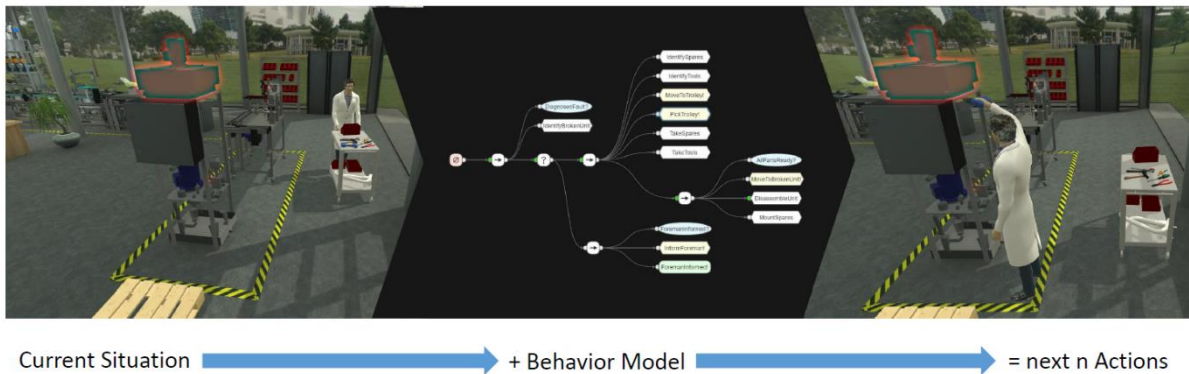
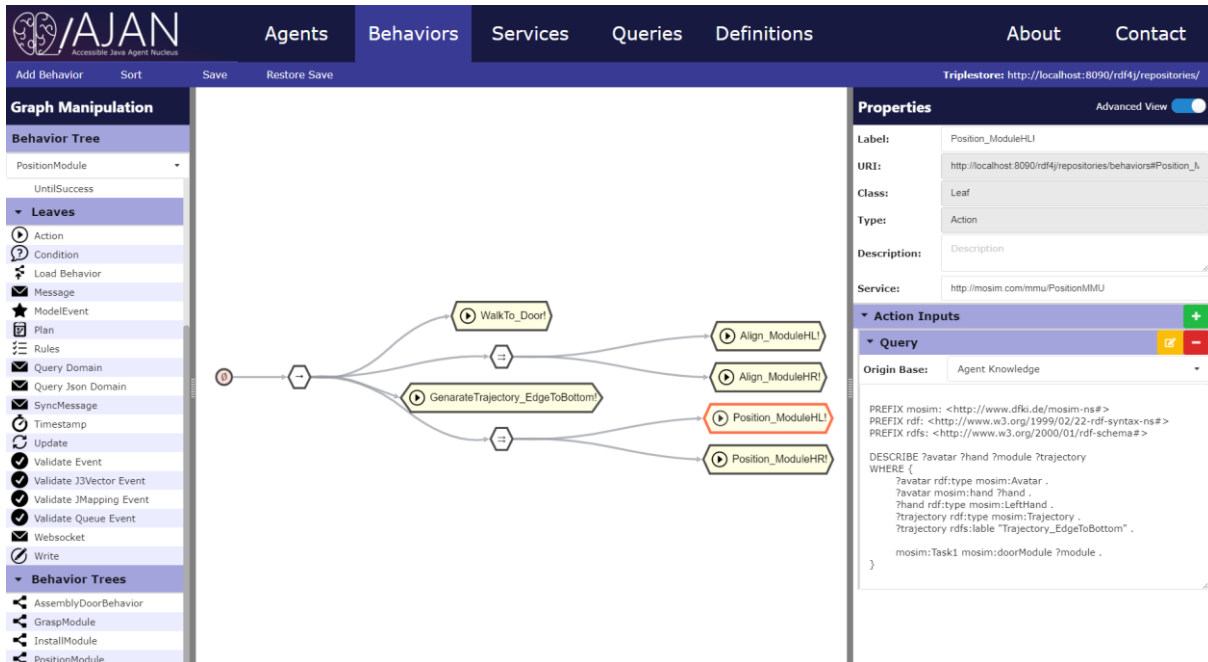


Figure 4 Reasoning Cycle of the proposed Behavior Modeling & Execution.

The BT paradigm is often used in combination with a blackboard that stores the current entity state. BTs can be event-based or executed with a game tick and are suitable for describing concurrent actions, fulfilling a primary criterion for selecting a behavioral model in MOSIM. However, they only carry out the actions that are ready for execution in the current situation. A generation of a list with possible future actions, as required by the co-simulator, is not yet possible with common implementations and will be thus investigated in the MOSIM project.

In the MOSIM project BTs are supposed to be used as an executable behavior model. The agent system AJAN developed at DFKI, was suggested as a possible implementation of the Behavior Modeling & Execution unit. In [9] and [10], the AJAN was presented and used for the control of simulated workers in an assembly line production. AJAN is a multi-agent system that is implemented as a Web service and uses BTs extended with SPARQL queries and RDF-Triple Stores as knowledge base. Furthermore, it is equipped with a graphical editor (see Figure 5) to model and manage agents with such BTs.



Properties

Label: Position_ModuleHL
 URI: http://localhost:8090/rd4j/repositories/behaviors#Position_M
 Class: Leaf
 Type: Action
 Description: Description
 Service: http://mosim.com/mmu/PositionMMU

Query

```

PREFIX mosim: <http://www.dfki.de/mosim-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

DESCRIBE ?avatar ?hand ?module ?trajectory
WHERE {
  ?avatar rdf:type mosim:Avatar .
  ?avatar mosim:hand ?hand .
  ?hand rdf:type mosim:LeftHand .
  ?trajectory rdf:type mosim:Trajectory .
  ?trajectory rdfs:label "Trajectory_EdgeToBottom" .
  mosim:Task1 mosim:doorModule ?module .
}
  
```

Figure 5 AJAN-Editor for agent and behavior modeling.

3 Technical Architecture

In order to combine several heterogeneous digital human simulation approaches in practice, a technical architecture fulfilling the requirements of the different use-cases and technology provider is necessary. In the following, a detailed presentation of the derived technical architecture for the MMI framework is provided.

3.1 Overview

The proposed framework comprises several components interconnected by a platform independent middleware. Figure 6 visualizes the overall framework and the involved components. In the following, the individual components are described in detail.

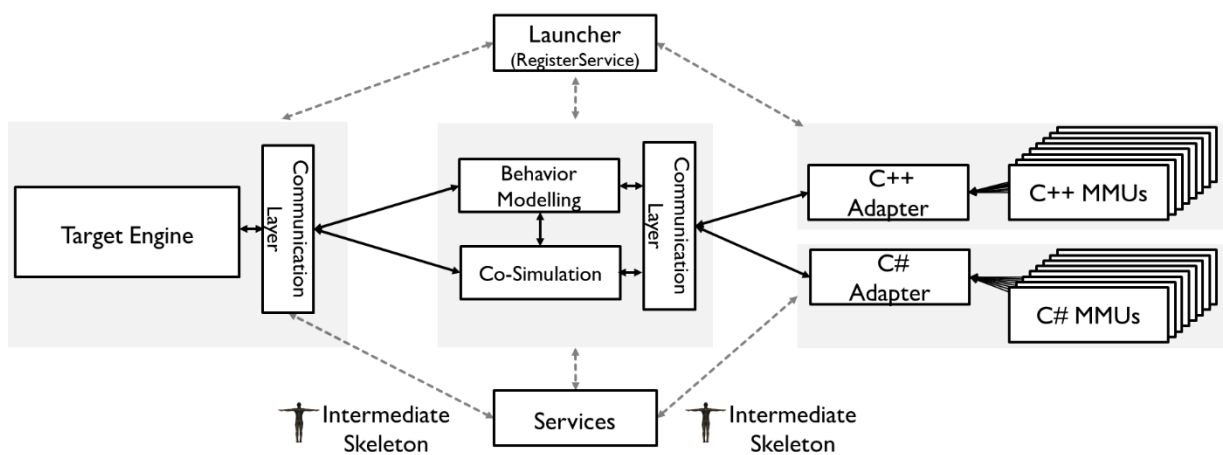




Figure 6 Overview of the proposed technical MOSIM architecture.

3.1.1 Motion Model Unit

To incorporate heterogeneous motion synthesis approaches in a common framework, it is indispensable to provide compatibility for many platforms. Therefore, each MMU can be realized in the particular programming language that best supports its approach for synthesizing motions. Each MMU implements the platform-independent interface as illustrated in Figure 2. The suggested interface is limited in its functionality to ease the cross-platform implementation. Additionally, each MMU provides a description file in which the motion type, specific parameters and the name of the unit are specified. To enlarge the possible portfolio of compatible algorithms, the MMUs can furthermore comprise a specific skeleton as required by the respective approach. The retargeting from/to the intermediate skeleton defined for the use in MOSIM (see Figure 6) can be performed by a dedicated retargeting service, which is available to the MMU. The MMUs themselves have a programming language specific format: Utilizing languages such as C#, C++, or Java, the MMUs are represented as .dll- or .jar files and can be instantiated at runtime. It is important to note, that co-simulation and MMU have an identical interface in order to allow nested MMUs. The MMU specific interface is described in Section 3.2. In general, independent of the programming language an MMU is represented as zip archive, which contains a description file and the programming language specific files/binaries.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

3.1.2 Adapter

If the MMUs are realized as separate standalone applications, each MMU must implement the communication by itself, which induces potential error sources and an implementation overhead. For instance, if the communication formats change over time (e.g., a new version of the standard is introduced), each MMU must be manually adjusted, which induces uneconomically high implementation efforts. Moreover, MMUs being realized as separately hosted applications might lead to performance bottlenecks, since the scene has to be synchronized multiple times for each MMU. To avoid this and counterbalance the drawbacks, the so-called adapters are proposed.

Adapters implement the communication protocols and are responsible for managing the MMUs, i.e., adapters act as proxy for the communication. In particular, adapters are provided for each compatible programming language (e.g. C#, C++, Java, Python). The components contain a session handling to allow multiple avatars and consumers using the same adapter instance. Moreover, adapters buffer the scene, while only the deltas are transmitted. Given the buffered scene, the MMUs contained in the adapter can access the scene with very low latency, since no further Web- or inter-process communications needs to be carried out. The adapter further provides access to the available services of the framework. A detailed overview of the specific interface of the adapter is available in Section 3.2.

3.1.3 Target Engine

As central accessing point of the end-user and the different use-cases, the target engine is responsible for the visualization of the scene and the digital human model. The component provides the ground truth scene and is realized in a specific programming language.

Additionally, an internal skeleton can be utilized within the target engine, whereas the retargeting from/to the reference skeleton can be semi-automatically performed by a retargeting service. The target engine needs to provide a standardized accessing functionality of the scene (e.g. get object with name "x", get transform of "y"). Furthermore, manipulations of the scene as intended by MMUs or the co-simulation must be applicable to the scene.



3.1.4 Communication Layer

Closely linked to the target engine, the so-called communication layer encapsulates the communication to the MMUs. The layer provides the accessing functionalities of the MMUs as they were on the local machine. The abstraction layer should be provided as implementation for each programming language, whereas the functionality might be adapted to specific applications.

3.1.5 Middleware

As an essential component for connecting the different components of the framework, the middleware is a crucial aspect. In general, vast amounts of different middleware solutions are available. For the desired MOSIM framework, different state of the art solutions have been examined. To ease cross-platform implementation, Apache Thrift [9] is utilized. Reasons for the selection of Apache Thrift for the planned standard are the availability as open source project, and the definition of the interfaces using a single source represented as Interface Definition Language (IDL) file. Moreover, the compatibility to the major programming languages and the performance were further major aspects that led to the selection of Apache Thrift.

The overall communication-formats and services of the framework are defined using the Thrift interface description language, whereas the source-code for the different platforms can be

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

automatically generated. In addition to the proposed document, the Thrift IDL files describing the MOSIM framework are provided as additional files online with the deliverable.

3.1.6 Services

To further increase the usability and shrink implementation efforts, functionality, which is oftentimes required by MMUs, is provided by so-called services. In particular, methods such as path planning, inverse kinematics, retargeting and collision-detection are offered as service. The services are realized using the common Apache Thrift middleware and implemented in a stateless manner, providing the capability to be used by multiple consumers. A detailed overview of the provided services in the MOSIM framework is given in Section 3.2.

3.1.7 Launcher

Despite the previously mentioned components, the novel framework additionally comprises a component that starts the adapters/services and provides a register for managing the respective component in the framework. The so-called launcher hosts a service, which acts as an entry point for the different components in the framework. In particular, the available adapters, services, as well as the respective connection information are provided. Moreover, the adapters and services actively register at the registry service. In this way, the users of the MOSIM framework only need the address information of this centralized service in order to gather all required information being necessary for the utilization of the framework. A detailed overview of the Launcher interface and the workflow is given in Section 3.2 and 3.3. The launcher application is provided in addition to this document as part of the MOSIM deliverable.

3.1.8 Intermediate Skeleton

Event though not depicting a separate component, the so-called intermediate skeleton is an important concept of the proposed framework. The intermediate skeleton is a standardized skeleton hierarchy/structure, which is utilized to transfer the posture data between the different components. Therefore, each avatar/skeleton must be mapped to the intermediate skeleton. A detailed overview of the specific format is given in Section 3.2.5.

3.2 Interfaces & Formats

After having introduced the overall architecture and the individual components, next, it is crucial to define the specific interfaces and formats used within the MMI framework. Thus, in the following the derived interfaces and the utilized formats of the MMI framework are revisited in detail (top-down approach).

3.2.1 Motion Model Unit

The **Motion Model Units** form the basic element in which the actual simulation approach is embedded. For usage across multiple programming languages and technologies, a unified interface, which fulfills the requirements of these heterogeneous approaches, is necessary. Figure 7 shows the basic interface as an UML class diagram. Furthermore, a detailed description of the available parameters and functions is given below.

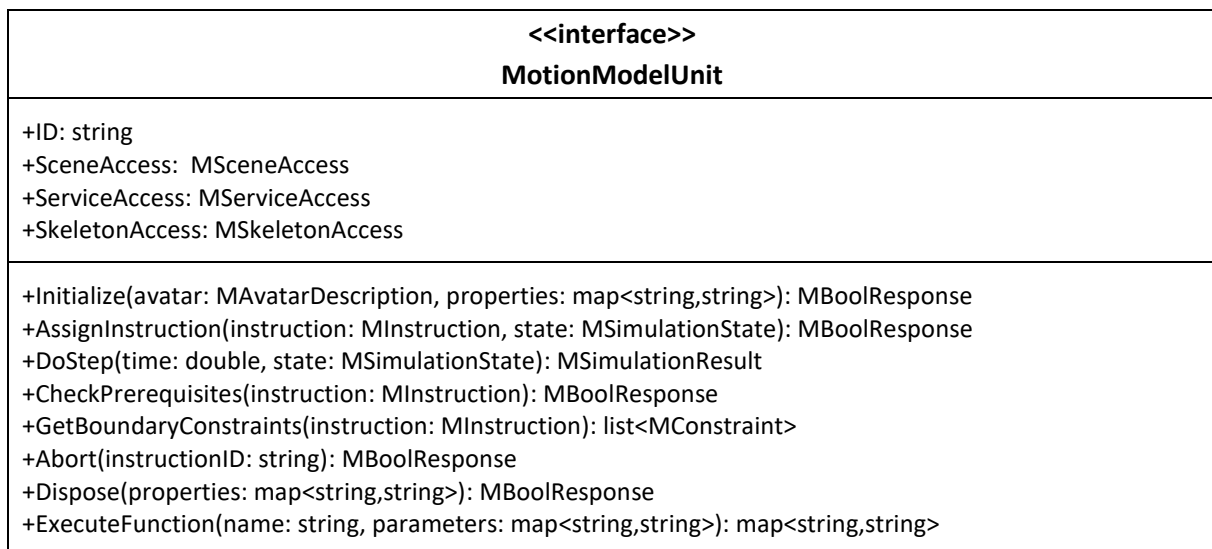


Figure 7 UML class diagram of the proposed Motion Model Unit interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	Unique id of the MMU. The ID is defined by the MMU developer during the MMU development process.
SceneAccess	Instance provides access to the scene (automatically set by the adapter).
ServiceAccess	Instance provides access to the services (automatically set by the adapter).
SkeletonAccess	Instance provides access to the helper functions of the avatar (automatically set by the adapter).

Initialize	Method to initialize the MMU. The function provides the description of the utilized avatar and additional properties as input. Within the function, the internal model of the MMU can be setup. The MMU needs to return the information whether the initialization was successful.
AssignInstruction	The method is used to assign a specific MInstruction to the MMU. Within the function, the MMU can compute suitable motions or evaluate the internal model.
DoStep	The DoStep method is called once a frame in order to trigger the computation of a new posture. In particular, the simulation frame time, as well as the current simulation state (MSimulationState) are provided as input. The MMU returns the computed results (MSimulationResult).
CheckPrerequisites	Based on a given MInstruction and the present state the MMU internally checks whether the instruction can be executed.
GetBoundaryConstraints	The method returns the boundary constraints for executing the MMU. In particular, these constraints can contain a full body posture. For instance, a data-driven MMU might return the first posture as start-boundary condition. The previous MMU can adjust the internal model to achieve the desired starting posture.
Abort	The method aborts the specified instruction.
Dispose	The Method disposes the full MMU including all ongoing computations.
ExecuteFunction	Debug functionality to call additional functions. This method should be only used for debugging purposes.

In particular, the Motion Model Unit comprises eight different functions, which must be provided by all implementations. Via the **Initialize** function, the unit is set up given the characteristics of the utilized avatar. These characteristics are provided within the **MAvatarDescription** class. Within the given class, the neutral posture of the utilized (intermediate) avatar is contained, as well as additional anthropometric values. This information can be utilized to adjust the internal model and skeleton to the used reference skeleton in the MMI framework. Furthermore, a map of additional properties can be specified for the initialization. These properties can be used for specification of additional initialization parameters. The MMU returns a Boolean flag (**MBoolResponse**) indicating whether the initialization was successful.

To define the instruction/motions to be computed by the MMUs, the **AssignInstruction** method is used. Within this function, a **MInstruction**, which is strongly inspired by the BML language, is utilized (see 3.2.4). This class type allows to formulate specific instruction such as “pick up object 1”, being performed by the MMU. Additionally, the present state of the Avatar/simulation (**MSimulationState**)

is used as input for the method. The method returns a Boolean status flag (**MBoolResponse**) that indicates whether the assignment was successful.

The main function for providing and generating the actual motion is the **DoStep** method. This function takes as an input the relative time to be simulated (e.g. 30 ms), as well as the current state of the simulation (**MSimulationState**). The output is the actual computed posture, actions and notifications for the given frame. In particular, the output is represented as **MSimulationResult** and contains further information. Despite the computed posture, the class also offers to specify posture constraints such as “fix position of left hand at (1, 1, 1)”. A detailed overview of all available constraints is given in the **MConstraint** section 3.2.4.3. Moreover, each MMU can return events that are an essential aspect for the co-simulation and the further workflow. Each MMU must provide an event of type end, if the last frame of the motion has been reached, and the MMU is finished. The list of the available events as well as the utilized **MSimulationEvent** class are described in 3.2.4.6. Additionally, each MMU can return proposals for manipulating the scene by using the **MSceneManipulation** class. For further debugging, it is possible to use **MDrawingCalls** for visualization and **LogData**.

For determining whether an instruction can be executed given the present state of the scene and the avatar the **CheckPrerequisites** method is provided. Each MMU can implement this method using internal starting criteria such as “left arm in range” for grasping.

To allow a simplified transition modeling between consecutive MMUs, each MMU can specify additional boundary constraints via the **GetBoundaryConstraints** method. A boundary constraint could be for instance the start posture of the generated motion. Therewith, the previous MMU can adjust the generated motion with respect to the given transition constraint.

To terminate a currently active instruction the **Abort** method is used. The method resets the MMU to the initial state before the assign instruction took place.

To terminate a given MMU, the **Dispose** method is utilized. In particular, the allocated resources within the MMU are cleared.

Finally, to increase the flexibility of the provided framework, a further method named **ExecuteFunction** is suggested. This function can be used for debugging or for future extensions of the framework.

Description file for Motion Model Units

To allow the utilization of the MMUs across multiple platforms, each MMU must provide a unique description file in analogy to the FMI standard. The description file contains relevant information such as the name of the MMU, information for the loading process as well as the motion type. Moreover, the parameters of the MMU as required for specifying a particular instruction are listed in the description file (e.g., one MMU might have a target parameter that is required, whereas another MMU has an optional velocity parameter). The UML class diagram of the MMU description file is visualized in Figure 8, whereas a detailed description of the parameters is listed below. Moreover, an exemplary description file for a MMU is provided in the Appendix.

MMUDescription

```



+Name: string
+ID: string
+AssemblyName: string
+MotionType: string
+Language: string
+Author: string
+Version: string
+LongDescription: string
+ShortDescription: string
+SupportedProportions: map<string,double>
+Properties: map<string,string>
+Dependencies: list<string>
+Events: list<string>
+Parameters: list<MParameter>

```

Figure 8 UML class diagram of the MMUDescription class, which is used for describing a specific MMU.

Detailed description of available parameters:

Parameter Name	Required	Description
Name	x	Name of the MMU (e.g. "WalkMMU").
ID	x	Unique id of the MMU.
AssemblyName	x	The name of the assembly to be loaded. This information is important for the adapters to load and instantiate the MMUs (e.g. walkMMU.dll).
MotionType	x	The motion types are utilized to express the provided motion and important for accessing the MMUs (e.g. "walk", "grasp", "crouch"). The motion type must be specified by the MMU developer.
Language	x	The programming language of the MMU (e.g. C#, C++). This information is important for the adapters.
Author	x	The author of the deployed MMU.
Version	x	The version of the deployed MMU.
LongDescription	x	A longer description of the MMU.
ShortDescription	x	The short description of the MMU.
SupportedProportions		The supported proportions/anthropometry of the MMU. It might be the case that an MMU is only working in a particular range.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Properties		Additional properties that can be set. Depending on the future functionalities of the framework aspects such as the requirement of low-level scene access can be specified here.
Dependencies		Dependencies, which need to be loaded by the adapter to instantiate the MMU.
Events	x	A list of all event types which are provided by the MMU (e.g. "start", "end", "foot_contactLeft", "foot_contactRight" .
Parameters	x	All available parameters, which can be specified for the MMU. The parameters are specified using the MParameter struct. For instance a MMU could provide parameters such as: {Name ="Target", Type="string", Required =true}, {Name ="Velocity", Type="double", Required =false}.

MMU File representation

Similarly, to the FMI standard, each MMU is represented as a zip archive on the file system. The zip archive contains the programming language independent description file (**MMUDescription**), as well as the programming language specific files required to instantiate the MMU. Using languages such as C# or C++, these files are represented as .dll files, whereas for Java .jar and for Python .py files are used.

3.2.2 Co-Simulation

As outlined in the previous chapter, the main responsibility of the co-simulation is to orchestrate multiple MMUs based on a list of specified instructions (**MInstruction**). In particular, the co-simulation needs to schedule the MMUs and incorporate the provided results. Analogously to the MMUs, a common interface for the co-simulation is proposed in the following. Figure 9 gives an overview of the derived interface, whereas below a detailed description is provided.

<<interface>> MCoSimulation
+ID: string +SceneAccess: MSceneAccess +ServiceAccess: MServiceAccess +SkeletonAccess: MSkeletonAccess

```



+Initialize(avatar: MAvatarDescription, mmus: list<string>, priorities: map<string,double>): MBoolResponse
+Initialize(avatar: MAvatarDescription, properties: map<string,string>): MBoolResponse
+AssignInstruction(instruction: MInstruction, state: MSimulationState): MBoolResponse
+DoStep(time: double, state: MSimulationState)
+CheckPrerequisites(instruction: MInstruction)
+GetBoundaryConstraints(instruction: MInstruction)
+Abort(instructionID: string)
+Dispose(properties: map<string,string>)
+ExecuteFunction(name: string, parameters: map<string,string>): map<string,string>

```

Figure 9 UML class diagram of the proposed co-simulation.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	Unique id of the Co-Simulation instance.
SceneAccess	Instance provides access to the scene (automatically set by the adapter).
ServiceAccess	Instance provides access to the services (automatically set by the adapter).
SkeletonAccess	Instance provides access to the helper functions of the avatar (automatically set by the adapter).
Initialize	Method is used to initialize the Co-Simulation. The function provides the description of the utilized avatar and additional properties as input. The co-simulation needs to return the information whether the initialization was successful. Specifically, for the co-simulation the utilized MMUs and the priorities needs to be provided. For compatibility with the MMU interface, the initialize function with the same signature as the MMU interface must be provided.
AssignInstruction	The method is used to assign a specific MInstruction to the co-simulation. Note that, the MInstruction class can contain a list of multiple instructions. In this way, a set of different tasks can be assigned at once.
DoStep	The DoStep method is called once a frame in order to trigger the computation of a new posture. In particular, the simulation frame time, as well as the current simulation state (MSimulationState) are provided as input. The co-simulation returns the computed results (MSimulationResult).
CheckPrerequisites	Method is provided for compatibility with the MotionModelUnit interface.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

GetBoundaryConstraints	Method is provided for compatibility with the MotionModelUnit interface.
Abort	The method aborts a specified instruction.
Dispose	The method disposes the co-simulation including all ongoing computations.
ExecuteFunction	Debug functionality to call additional functions. This method should be only used for debugging purposes.

The interface is strongly related to the interface of the **Motion Model Unit**. A reason for this is, that nested MMUs that co-simulate other MMUs must be allowed within the framework. Therefore, the co-simulation must be accessible in an identical way as the MMU. The main difference between the co-simulation and MMUs are with regard to the initialization. For the co-simulation the list of the utilized MMUs as well as the priorities, need to be additionally provided as input. The overall task of the co-simulation is to coordinate multiple MMUs and instructions. To allow the specification of multiple instructions, the **MInstruction** class can contain a list of further instructions. Even though the interface is strongly similar to the interface of the MMU, depending on the utilized environment, the co-simulation can be extended with custom functionalities (e.g., w.r.t. behavior modeling) or directly embedded in the target engine using an entirely different interface.

3.2.3 Behavior Modeling & Execution

The Behavior Model and Execution unit (**MBehaviorExecution**) generates instructions (**MInstruction**) for the co-simulator based on the scene and a predefined behavior model. The entity that generates these instructions for an individual simulated character and manages its knowledge is called agent, which is represented with the **MAgent** class. **MBehaviorExecution** manages these agents. With the Behavior Modelling & Execution unit, so-called agent templates (**MTemplate**) can be created, which describe a certain type of agent. Such a template contains configuration information about an agent and its behavior models. It is a blueprint for an agent (**MAgent**), so to speak. To create an agent, the name of the agent and the template to be used as well as its initial knowledge (**MInitialKnowledge**) or the initial state of the scene to be viewed are transferred to the Behavior Model & Execution unit.

MBehaviorExecution
+Agents: list<MAgent> +Templates: list<MTemplate>

```

+AddTemplate(MTemplate):bool
+DeleteTemplate(MTemplate):bool
+CreateAgent(template:ID, name, MInitialKnowledge):MAgentState
+ListAgents():list<MAgentState>
+DeleteAgent(agent:ID):list<MAgentState>

```

Figure 10 UML class diagram of the MBehaviorExecution class.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
Agents	List of all available agents
Templates	List of all available agent templates
AddTemplate	Add new agent template which describes an agent model
DeleteTemplate	Delete an agent template
CreateAgent	Create an agent with a defined name, linked agent template and initial agent knowledge
ListAgents	List all agents
DeleteAgent	Delete specified agent
Agents	List of all available agents
Templates	List of all available agent templates
AddTemplate	Add new agent template which describes an agent model

In addition to its behavior, an agent has several storage units for storing RDF-based data, the so-called Triple Data Bases (**MTripleDataBase**). In these the scene updates (**MSceneUpdate**) are stored which are mapped to RDF in an intermediate step. These triple stores are accessed by the behaviors (**MBehavior**) during their execution in order to query states or to read data for the generation of instructions (**MInstruction**). Actions or MMUs and services that should be available for the behavior models will be referenced in the agent.

MAgent
+ID: string +Name: string +SceneAccess: MSceneAccess +ServiceAccess: MServiceAccess +AgentState: MAgentState +Beliefs: list<MTripleDataBase>

+Behaviors: list<MBehavior> +Services: list<MService:ID> +Actions: list<MMU:ID>
+Initialize(list<MSceneUpdate>):MAgentState +GetState():MAgentState +Perceive(list<MSceneUpdate>):bool +CompleteInstruction(MInstructionResult):bool +Run(list<MSceneUpdate>,<MBehavior>):MInstruction +Abort(<MBehavior>):bool

Figure 11 Class diagram of the MAgent class.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	Agent ID.
Name	Name of the Agent.
SceneAccess	Access to the Scene for agent-scene interaction, like reading out the current state of the scene or to manipulate it.
ServiceAccess	Access to available services to make use of their provided methods while behavior execution.
AgentState	State of the agent.
Beliefs	All beliefs of an agent stored as RDF in multiple triple stores.
Behaviors	List of all SPARQL-BTs of the agent.
Services	List of used services.
Actions	List of used MMUs.
Initialize	Method to initialize the agent by executing a specific SPARQL-BT for configuration purposes, like setting variables.
GetStatus	Method for returning the agent status.
Perceive	Method which is used by the consumer to update the agent knowledge if a scene or simulation update comes up:
CompleteInstruction	Method to tell the agent that an action or motion instruction is finished successfully or faulty.
Run	Method to run the defined SPARQL-BT of the agent.

Abort	Method to abort a running SPARQL-BT.
--------------	--------------------------------------

In the tree-based Behavior Model (**MBehavior**), which is defined as a Behavior Tree, five important classes of leaf nodes (**MBTLeafNode** \leftarrow **MBTTreeNode**) or executable primitives will be used: **MBTCondition**; **MBTUpdate**; **MBTService**; **MBTAction**; **MBTScene**. All these nodes are united by the fact that they access the knowledge base of the agent via SPARQL queries and process the resulting information or forward it to a service, for example. With so-called conditions (**MBTCondition**) the current state of the simulation or the character to be simulated is queried; with updates (**MBTUpdate**) the internal knowledge of the agent can be updated; services nodes (**MBTService**), with which external or internal services are called to receive external data or to process internal data; and action nodes (**MBTAction**), which represent the individual instruction steps or MMUs. In addition, further types of nodes (**MBTBranchNode** \leftarrow **MBTTreeNode**) are used to define the internal operational logic of the BT. With such a node the functionality mentioned in the previous chapter to generate instructions (**MInstruction**) based on linked actions and tree nodes, will be implemented. At each execution cycle of the agent, such an instruction is generated and sent to the co-simulator for execution. Successful or failed instruction execution is reported (**MInstructionResult**) from the co-simulator to the agent.

MBTLeafNode and **MBTBranchNode** are derived from **MBTTreeNode**. They are not considered in the following list of classes and interfaces, since they are represented by their relevant variants for the project. To summarize, a branch node has at least one child node and all leaf nodes have queries with which they access the knowledge base of the agent.

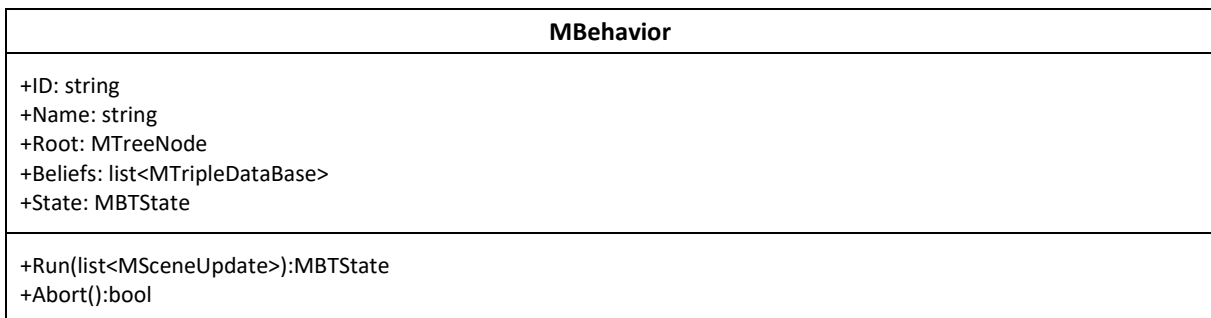


Figure 12 UML class diagram of the MBehavior class.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the behavior
Name	Name of the behavior
Root	Root node or starting point of the behavior
Beliefs	Link to the agent beliefs

State	State of the behavior: FRESH, RUNNING, SUCCESS, FAILURE, ABORTED
Run	Method to run the behavior
Abort	Method to abort the behavior

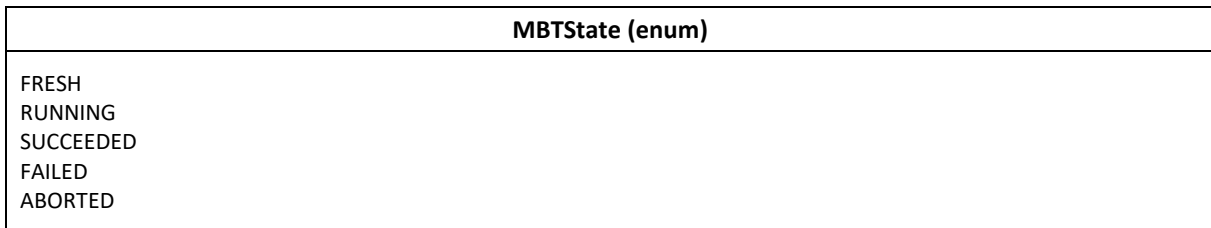


Figure 13 UML class diagram of the MBTState enum.

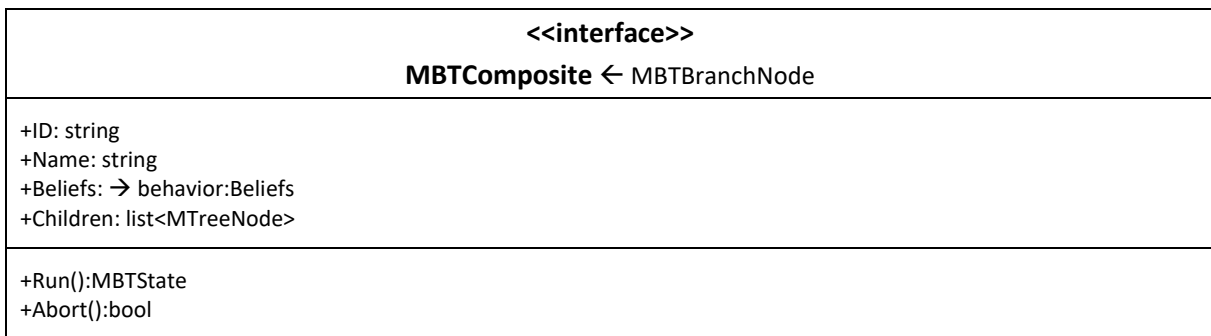




Figure 14 Overview of the MBTComposite interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the composite node.
Name	Name of the composite node.
Beliefs	Beliefs of the agent.
Children	Children of the composite node.
Run	Method to run the composite executed by its parent node, returning its status.
Abort	Method to abort composite node.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

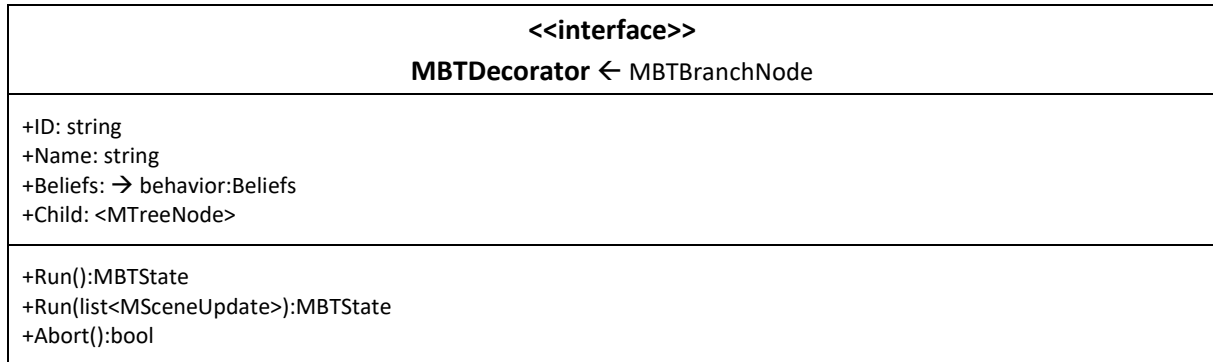
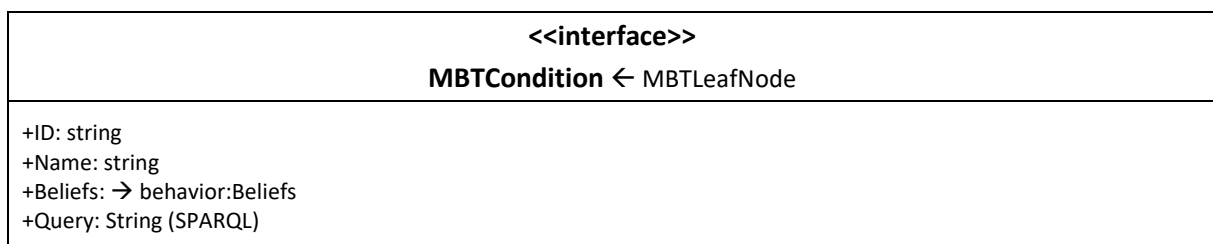




Figure 15 Overview of the MBTDecorator interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the decorator node.
Name	Name of the decorator node.
Beliefs	Beliefs of the agent.
Child	Child of the decorator node.
Run	Method to run the decorator node executed by its parent node, returning its status.
Abort	Method to abort the decorator node.

With composite nodes (**MBTComposite**) that can have several child nodes, sequential or parallel processes are usually implemented, whereas decorators (**MBTDecorator**) possess only one child node. This type is mostly used for loops. In contrast to the tree nodes just mentioned, Leaf Nodes (**MBTLeafNode**) do not have child nodes.



	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

+Run():MBTState

Figure 16 Overview of the MBTCondition interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the leaf node.
Name	Name of the leaf node.
Beliefs	Beliefs of the agent.
Query	ASK SPARQL query which defines an agent state.
Run	Method to execute this node, which performs the defined query on the agent beliefs, returning: SUCCEEDED or FAILED

Example:

MBTCondition	
ID	“:WalkToTarget1?” (RDF)
Name	“WalkToTarget1?”
Beliefs	“ajan:AgentKnowledge”
Query	<pre> “PREFIX mosim: <http://mosim/vocabulary> ASK WHERE { ?avatar a mosim:Avatar . ?avatar mosim:locatedNextTo ?location . ?target a mosim:Target . ?target mosim:name “Target1” . FILTER (?location != ?target) }” </pre>

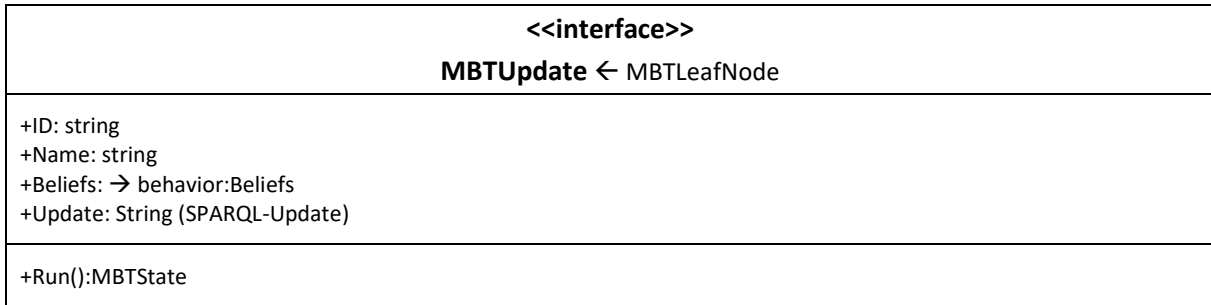




Figure 17 Overview of the MBTUpdate interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the leaf node.
Name	Name of the leaf node.
Beliefs	Beliefs of the agent.
Update	UPDATE SPARQL query which defines a new agent state.
Run	Method to execute this node, which performs the defined query on the agent beliefs, returning: SUCCEEDED or FAILED

Example:

MBTUpdate UpdateAvatarLocation	
ID	":UpdateAvatarLocation" (RDF)
Name	"UpdateAvatarLocation?!"
Beliefs	"ajan:AgentKnowledge"
Update	<pre> PREFIX mosim: <http://mosim/vocabulary> DELETE {?avatar mosim:locatedNextTo ?location .} INSERT {?avatar mosim:locatedNextTo ?target .} WHERE { ?avatar a mosim:Avatar . ?avatar mosim:locatedNextTo ?location . ?target a mosim:Target . </pre>

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

```

?target mosim:name "Target1" .
FILTER (?location != ?target)
}

```



<<interface>> MBTService ← MBTLeafNode
+ID: string +Name: string +Beliefs: → behavior:Beliefs +Query: String (SPARQL-Update) +Service: MService:ID
+Run():MBTState

Figure 18 Overview of the MBTService interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the leaf node.
Name	Name of the leaf node.
Beliefs	Beliefs of the agent.
Query	CONSTRUCT or SELECT SPARQL query, which defines an agent state.
Service	Link to the service, which has to be used.
Run	Method to execute this node, which performs the defined query to gather data from the agent beliefs which will be sent to the defined service, returning: SUCCEEDED or FAILED or RUNNING

MBTScene ← MBTLeafNode
+ID: string +Name: string +Beliefs: → behavior:Beliefs +Query: String (SPARQL-Update) +Scene: MScene:ID

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

+Run():MBTState

Figure 19 Overview of the MBTScene interface

Detailed description of available parameters/functions:



Function/ Parameter Name	Description
ID	ID of the leaf node.
Name	Name of the leaf node.
Beliefs	Beliefs of the agent.
Query	UPDATE SPARQL query which defines an agent state.
Scene	Link to the scene.
Run	Method to run this node, which executes the MScene: GetChanges() method and performs the defined query to update the agent beliefs with the received scene updates, returning: SUCCEDED or FAILED

<<interface>> MBTAction ← MBTLeafNode
+ID: string +Name: string +Beliefs: → behavior:Beliefs +Query: String (SPARQL-Update) +Action: MMUID
+Run():MBTState +Validate(MInstructionResult):bool

Figure 20 Overview of the MBTAction interface.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
ID	ID of the leaf node.
Name	Name of the leaf node.
Beliefs	Beliefs of the agent.
Query	CONSTRUCT or SELECT SPARQL query which defines a agent state.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Action	Link to the MMU which have to be used.
Run	Method to execute this node, which performs the defined query to gather data or MMU constraints from the agent beliefs which will be used by the co-simulator, returning: RUNNING
Validate	Method to validate the incoming MInstructionResult to decide if the action performed by the co-simulator was successful, returning: SUCCEEDED or FAILED

Example:

MBTAction Walk	
ID	“:WalkToTarget1” (RDF)
Name	“WalkToTarget1”
Beliefs	“ajan:AgentKnowledge”
Action	“walk” (MotionType)
Query	<pre> “PREFIX mosim: <http://mosim/vocabulary> SELECT ?target WHERE { ?target a mosim:Target . ?target mosim:name “Target1” . }” </pre>

3.2.4 Core Formats

The formats being described within this section build the core of the MMI framework and are relevant for the usage of the Motion Model Units, the Co-Simulation, as well as for the behavior modeling and the target engine.

3.2.4.1 Motion Instruction (MInstruction)

The so-called **MInstruction** class represents an important format in order to specify a desired motion. The format is strongly inspired by the behavior markup language (BML), see [10], [11]. Below, the class diagram as well as a detailed description of the parameters and practical examples are provided.

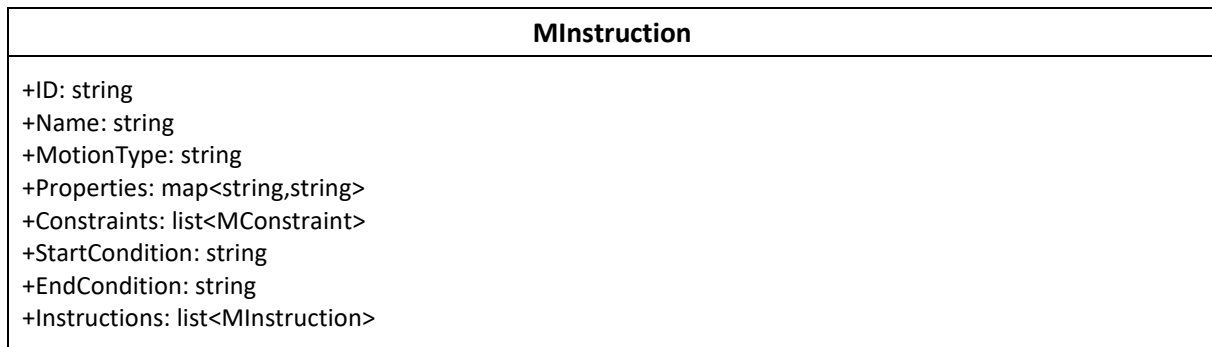


Figure 21 UML class diagram of the MInstruction class, which is utilized for specifying motion instructions for a given MMU.

Detailed description of available parameters:

Parameter Name	Required	Description
ID:	x	Represents a unique id, which is utilized to reference the motion instructions. The ID must be generated by the user/behavior, which create the instruction.
Name:	x	A name, which can be specified for the given instruction. The name does not need to be unique (e.g. "WalkInstruction1").
MotionType	x	The type is used to access the desired motion (e.g. walk, grasp). Each MMU has in additional also a specified motion type. By matching the motion types, a suitable MMU can be identified and accessed. The motion type must be specified by the user/behavior creating the instruction. In particular, the available motion types (provided by the MMUs) must be known a priori.
Properties		Optional dictionary which can contain properties for the MMUs (e.g. {"Velocity", "1.0"}, {"TargetObject", "Object1"}, {"MMU_ID", "xy"}).
Constraints		Constraints can be optionally set for the instruction. For instance, a constraint describing the end posture (=start posture of

		subsequent MMU) can be inserted. The different types of constraints in the framework are described in 3.2.4.3.
StartCondition		Relevant information for the timing of the motion within the co-simulation. The conditions are strongly inspired by BML: Example: id_task1:end (<bml start="id_task1:End"/>)
EndCondition		Relevant information for the timing of the motion. Given a end-condition a motion can be terminated after a specific event occurred or a specified amount of time is elapsed. Example: id_task1:end (<bml end="id_task1:End"/>)
Instructions		A MInstruction can contain additional MInstruction in order to represent tree structures utilized for hierarchical co-simulation or behavior modeling. Note for MMU utilization is it expected that only one instruction is utilized for each MMU.

Examples:

MInstruction Walk	
ID	"12345"
Name	"Walk Instruction 1"
MotionType	"walk"
Properties	{"Target", "Walk Target 1"}

MInstruction Grasp Object	
ID	"23456"
Name	"Grasp Instruction 1"
MotionType	"grasp"
Properties	{"Target", "Object1"} {"Hand", "Left"} {"Velocity", "1.0m/s"}
Start-condition	12345:End

3.2.4.2 Motion Instruction Result (MInstructionResult)

The so-called **MInstructionResult** represents the result or status of an executed **MInstruction**. The co-simulator sends it to the Behavior Execution, which can generate new **MInstructions** based on it. The **MInstructionResult** is sent to the Behavior Execution each time an instruction is executed by the co-simulator. In addition to the ID and the state of the executed **MInstruction**, this contains a list of further **MInstructionResults** as well as properties that contain possible reasons for a failure.

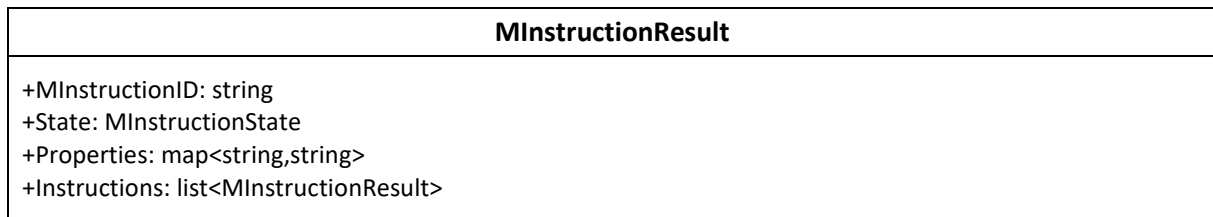


Figure 22 UML class diagram of the MInstructionResult class.

Detailed description of available parameters:

Parameter Name	Required	Description
MInstructionID	x	The ID of the corresponding instruction.
State	x	State of the corresponding instruction
Properties		Optional dictionary which can contain properties for the MMUs
Instructions		Optional list of linked MInstructions

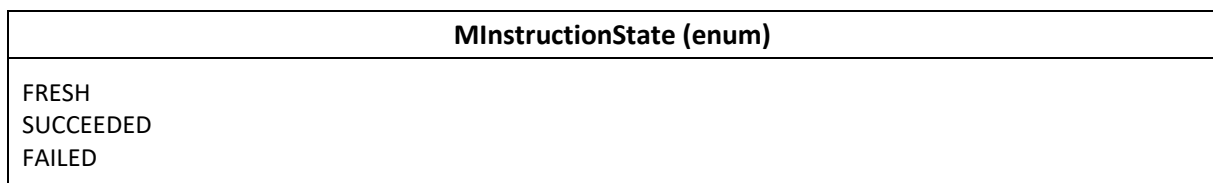


Figure 23 UML class diagram of the MInstructionState enum.

3.2.4.3 Constraints (MConstraint)

Besides the **MInstruction**, constraints are also an important concept within the MMI framework. The framework provides a base class named **MConstraint**. Since Apache Thrift does not allow class inheritance, the **MConstraint** class contains all available Constraint Types explicitly modeled.



```

+ID: string
+EndeffectorConstraint: MEndeffectorConstraint
+TrajectoryConstraint: MTrajectoryConstraint
+ParentingConstraint: MParentingConstraint
+PositionConstraint: MPositionConstraint
+RotationConstraint: MRotationConstraint
+JointAngleConstraint: MJointAngleConstraint
+PostureConstraint: MPostureConstraint
+Properties: map<string, string>

```

Figure 24 UML class diagram of the MConstraint class that is used for specifying a specific constraint in the framework.

Detailed description of available parameters:



Parameter Name	Required	Description
ID	x	Unique id of the constraint.
EndeffectorConstraint		A constraint that can specify a desired endeffector position/rotation represented as MEndeffectorConstraint .
TrajectoryConstraint		A constraint for specifying a trajectory using the MTrajectoryConstraint class.
ParentingConstraint		A constraint describing the parenting to specific object.
PositionConstraint		A constraint describing positional restrictions.
RotationConstraint		A constraint describing rotational restrictions.
PostureConstraint		An assigned posture constraint (if defined).
Properties		Optional properties, which can be specified.

An essential constraint type for the framework are the so-called **MPostureConstraints**. This constraint type can be utilized to describe a specific (sub) posture, which is required. Additionally, weights for the different bones, as well as velocity and acceleration information can be specified.

MPostureConstraint
<pre> +Posture: MAvatarPosture +Weight: map< MJointType, double> +Velocity: map<MJointType, list<double>> +Acceleration: map<MJointType, list<double>> </pre>

Figure 25 UML class diagram of the MPostureConstraint. The constraint type allows to specify a full posture as constraint.

Detailed description of available parameters:

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Parameter Name	Required	Description
Posture	x	The MAvatarPosture , which describes the desired posture.
Weight		An optional weight for each joint.
Velocity		An optional velocity for each joint (might be necessary for posture blending).
Acceleration		An optional acceleration for each joint (might be necessary for posture blending).

For specifying the position and rotation of endeffectors (e.g. left hand, right foot), the **MEndeffectorConstraint** class can be utilized. The constraint describes the specific endeffector type (e.g. left hand, right foot) and can contain optional position and rotation constraints. These constraints can be defined either as global coordinates or relative to another transform.

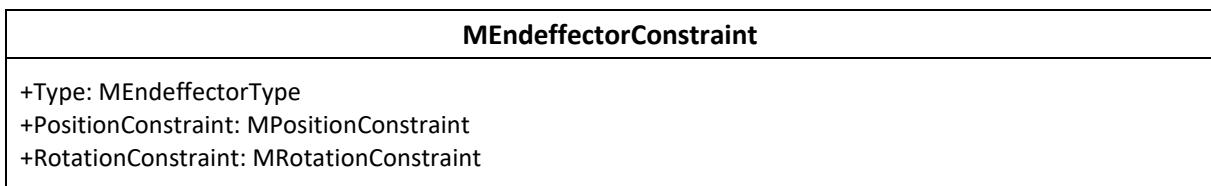
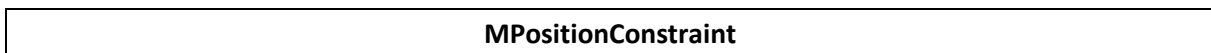


Figure 26 Overview of the UML class diagram of the MEndeffectorConstraint.

Detailed description of available parameters:

Parameter Name	Required	Description
Type	x	The specified endeffector type (e.g. left hand, right foot). The user/MMU intending to specify an endeffector constraint must specify the respective endeffector type from the MEndeffectorType enum.
PositionConstraint		The position constraint for the endeffector.
RotationConstraint		The rotation constraint for the endeffector.

To fix a position relative to a specific parent or globally the **MPositionConstraint** can be used. The position constraint can be set absolute (if no parent is defined) or relative to specific parent (if defined). Moreover, the respective values can constraint only a single axis or represent a full 3D position. The respective values can be interpreted by the MChannel values.



```

+Values: list<double>
+Channels: list<MChannel>
+Parent: string

```

Figure 27 UML class diagram of the MPositionConstraint.

Detailed description of available parameters:

Parameter Name	Required	Description
Values	x	The actual values of the constraint (e.g. list of positions in Cartesian coordinates).
Channels	x	The channels for the specified position constraint. Analogously to the MAvatarPosture, the channels can contain an arbitrary number of components ([1; 3]) therefore expressing also constraints which are only addressing specific axes.
Parent		An optional parent. The defined constraint is applied relative to the parent (if defined).

Analogously to the **MPositionConstraint**, in the MOSIM framework a rotation constraint is provided. Here again the individual values can restrict only a single axis or a full 3D rotation. The **MChannel** class can be used to interpret the data.

MRotationConstraint		
<pre> +Values: list<double> +Channels: list<MChannel> +Parent: string </pre>		

Figure 28 UML class diagram of the MRotationConstraint.

Detailed description of available parameters:

Parameter Name	Required	Description
Values	x	The actual values of the constraint (e.g. list of rotations in euler angles).
Channels	x	The channels for the specified rotation constraint. Analogously to the MAvatarPosture, the channels can contain an arbitrary number of components ([1;3]) therefore expressing also constraints which are only addressing specific axes.
Parent		An optional parent. The defined constraint is applied relative to the parent (if defined).

For describing a set multiple position and rotations constraints, the so-called **MTrajectoryConstraint** can be utilized. The trajectory constraints the object specified within the reference variable.

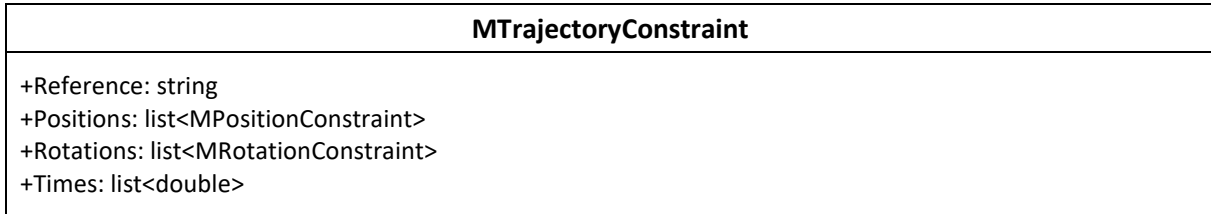


Figure 29 UML class diagram of the MTrajectoryConstraint that is used to describe restricting along a trajectory.

Detailed description of available parameters:

Parameter Name	Required	Description
Reference	x	The reference joint/avatar, which should be constrained using the given trajectory. The reference must be specified by the user (e.g., ID of an MSceneObject).
Positions	x	The position constraints of the trajectory.
Rotations	x	The rotation constraints of the trajectory.
Times		Optional timing constraints for the trajectory. If no timing dependencies are required, the field can be left empty (temporal realization depends on the actual instance that realizes the constraints).

To constraint an object relatively to a parent object, the **MParentingConstraint** can be utilized. In particular, the position and rotation offset to the parent can be specified.

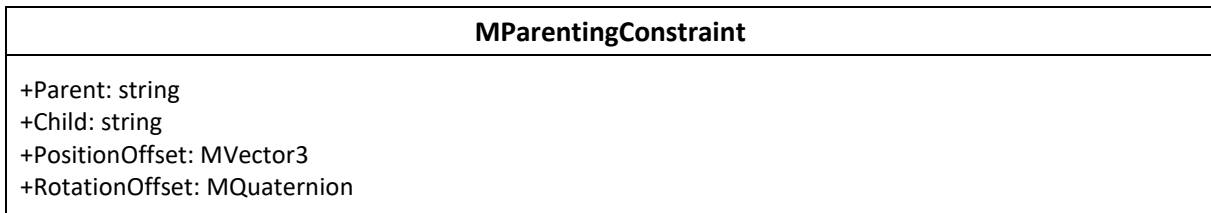




Figure 30 UML class diagram of the MParentingConstraint.

Detailed description of available parameters:

Parameter Name	Required	Description
Parent	x	The id of the parent object.
Child	x	The id of the child object.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

PositionOffset	x	The relative offset of the child.
RotationOffset	x	The relative rotation offset of the child.

3.2.4.4 Simulation State (MSimulationState)

The so-called **MSimulationState** class is an important format for calling MMUs. The class contains the actual simulation state in terms of posture of the avatar, presently active constraints, desired scene manipulations and events created by the MMUs in the current frame.

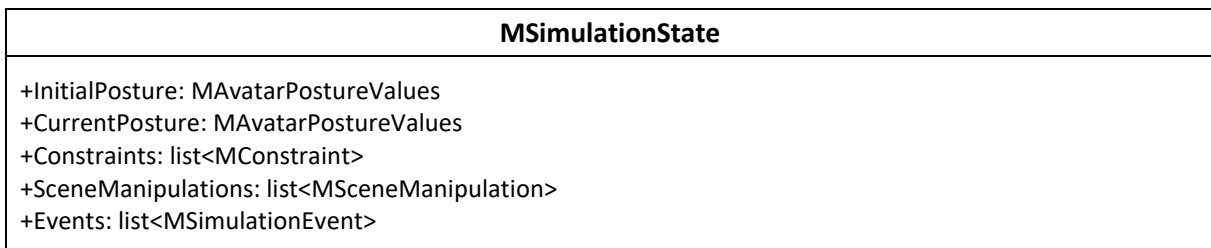


Figure 31 UML class diagram of the MSimulationState. The class is used as input for the MMUs, describing the current state of the Avatar, as well as current operations created by previous MMUs in hierarchy (e.g., constraints, events).

Detailed description of available parameters:

Parameter Name	Required	Description
InitialPosture	x	Values, which represent the approved and merged posture of the last frame. The variable describes a single posture.
CurrentPosture	x	Values, which represent the generated posture of the previously executed MMU in hierarchy in current frame. The variable describes a single posture. Depending on the co-simulation, this variable can contain either a merged posture or just the plain result of the previous MMU.
Constraints		A list of currently active constraints (e.g. left foot should stay at ground).
Scene Manipulations		A list of desired manipulations of the scene that can be optionally specified by the MMU. Using this approach contradicting scene manipulations can be avoided.
Events		A list of all events raised within the current frame. This also includes the MMUs raised by previous MMUs in the present frame (e.g. MMU:end). A detailed overview of the available events is given in the MSimulationEvent section.

3.2.4.5 Simulation Result (MSimulationResult)

Another essential class for MMU handling is the so-called **MSimulationResult**. The class contains the computed information returned by an MMU. In particular, information about the posture, constraints, events and scene-manipulations are provided. Given the format, it is furthermore possible to specify intended drawing calls being visualized in the target engine, as well as providing debug information using the LogData.

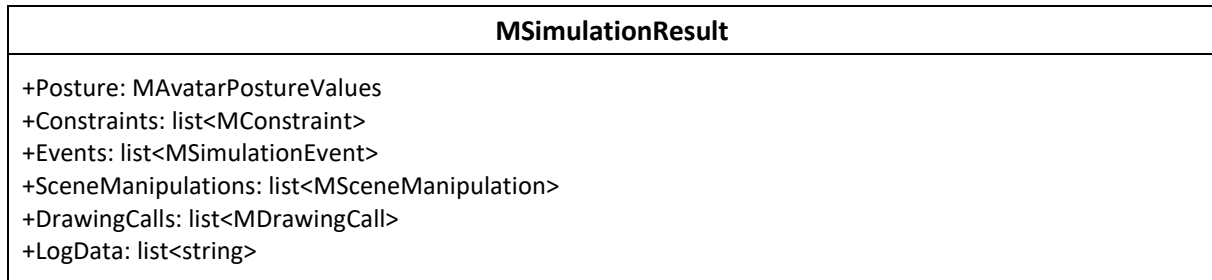


Figure 32 UML class diagram of the MSimulationResult. The class is returned by the MMU executing the DoStep routine.

Detailed description of available parameters:

Parameter Name	Required	Description
Posture	x	Values, which represent the computed posture of the MMU for the current frame. Instead of the full hierarchy, only the root transformation and the local joint rotations are transferred (see MAvatarPostureValues).
Constraints		The list of constraints which should be handled by consecutive MMUs/co-simulation (e.g. left foot should stay at ground).
Events		A list of status messages (called events) of the MMU. The framework comprises default events such as End, Start, and Abort similar to BML. The events are processed by the co-simulation.
Scene Manipulations		A list of desired manipulations of the scene that can be optionally specified by the MMU. Using this approach contradicting scene manipulations can be avoided.
Drawing Calls		Optional parameter to send visualization instructions to the target engine (e.g. draw the path [{1,2},{1,3}].
Log Data		Optional list of strings for providing debugging information.

3.2.4.6 Simulation Event (MSimulationEvent)

The so-called **MSimulationEvent** is an important class for the scheduling and orchestration of multiple MMUs. Similar to BML, an event is always referred to a **MInstruction**. This means, that each **MSimulationEvent** needs a corresponding **MInstruction**. Moreover, within the framework predefined MSimulationEvent Types are provided (again see bml). Each event can have an additional name, as well as further properties.

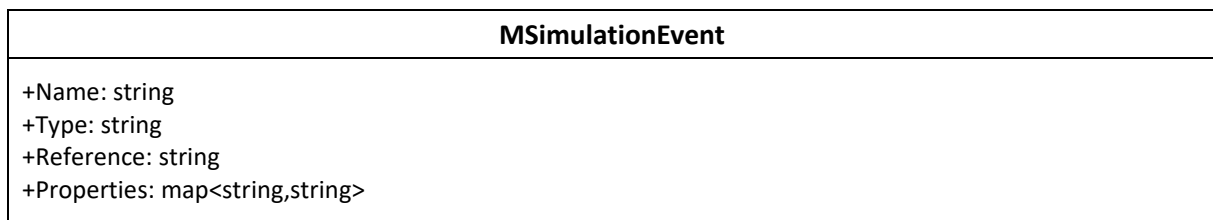


Figure 33 UML class diagram of the MSimulationEvent class. The class is utilized for representing an event, created by a MMU or co-simulation.

Detailed description of available parameters:

Parameter Name	Required	Description
Name	x	A descriptive name of the Event. Can be specified by the MMU internally.
Type	x	The type of the Event. The framework provides basic types such as End and Start. However, MMU developers can define their own events such as (Foot contact, Grasp contact)
Reference	x	The unique ID of the corresponding MInstruction.
Properties		Optional properties, which can be transmitted additionally.

Within the framework a basic set of events, inspired by the bml approach is provided. However, each MMU developer can furthermore specify custom events. The default list of available simulation events is illustrated below.

Default events provided in the framework:

Name (case sensitive)	Description
start	Event is automatically provided by the co-simulation if a MMU is started (see bml).
ready	Optional event (see bml).
stroke_start	Optional event (see bml).

stroke_end	Optional event (see bml).
end	Event must be provided by the MMU if the MMU is finished (see bml).
abort	Event if the present MMU is aborted.
warning	An optional warning event.
exception	Event should be raised if an exception occurred within the MMU.

3.2.4.7 Scene Manipulations (MSceneManipulation)

An important format for expressing manipulations of the scene is the **MSceneManipulation** class. The format is utilized by the MMUs to express intended scene manipulations (e.g. object is at position (x,y) and for the synchronization of the scene within the adapters. A detailed overview of the individual classes **MTransform** and **MSceneObject** is provided within the scene section.

MSceneManipulation
+Transforms: list<MTransformManipulation> +PhysicsInteractions: list<MPhysicsInteraction> +Properties: list<MPropertyManipulation>

Figure 34 UML class diagram of the MSceneManipulation class. This class is used to express manipulation, which should be applied to the scene (e.g., positional change of scene object, physical interaction).

Detailed description of available parameters:

Parameter Name	Required	Description
Transforms		List of intended manipulations of the transforms of specific object (e.g. change location of object 1).
PhysicsInteractions		List of intended physics interactions of given objects.
Properties		List of intended property changed (e.g. set flag of MSceneObject).

The **MTransformManipulation** class allows specifying manipulations of the location/transformation a specific object in the scene. The target of the transform manipulation must be always set, whereas the position, rotation and parenting can be optionally set.

MTransformManipulation

```

+Target: string
+Position: MVector3
+Rotation: MQuaternion
+Parent: string

```

Figure 35 UML class diagram of the MTransformManipulation.

Detailed description of available parameters:

Parameter Name	Required	Description
Target	x	The id of the target object/Avatar, where the manipulation should be carried out.
Position		The changed position. The values represent the new absolute position. If undefined it will be ignored.
Rotation		The changed rotation. The values represent the new absolute rotation. If undefined it will be ignored.
Parent		The new parent value. If undefined it will be ignored. The present parent is removed if flag Remove:id is used.

The **MPhysicsInteraction** class can be utilized to express intended physical interactions with the scene. In particular, a specific target (referenced by the ID) can be affected. The set of physical interaction types is specified by the **MPhysicsInteractionType** enum as shown below.

MPhysicsInteraction
<pre> +Target: string +Type: MPhysicsInteractionType +Values: list<double> +Properties: map<string,string> </pre>

Figure 36 UML class diagram of the MPhysicsInteraction class, used to express physical interactions with the scene (e.g. , apply force).

Detailed description of available parameters:

Parameter Name	Required	Description
Target	x	The target object at which the physics interaction should be applied (unique id).
Type	x	The specified MPhysicsInteractionType enum.
Values	x	The values which express the physics interaction (e.g. force, torque)

Properties	Optional properties which can be specified.
-------------------	---

For describing the specific type of a physics interaction, the **MPhysicsInteractionType** enum is provided in the framework.

MPhysicsInteractionType (enum)
AddForce AddTorque ChangeVelocity ChangeAngularVelocity ChangeMass ChangeCenterOfMass ChangeInertia

Figure 37 UML class diagram of the MPhysicsInteractionType enum. The enum describes all possible physical interactions in the framework.

The **MPropertyManipulation** class can be utilized to adjust given properties of an **MSceneObject** or **MAvatar**.

MPropertyManipulation
+Target: string +Key: string +Value: string

Figure 38 UML class diagram of the MPropertyManipulation class.



Detailed description of available parameters:

Parameter Name	Required	Description
Target	x	The id of the target object/Avatar, where the manipulation should be carried out.
Key	x	The key of the property.
Value		The new value of the property.

3.2.4.8 Further formats

Despite the previously introduced formats, the framework also comprised additional basic formats being described within the following:

One important format for calling functions with a Boolean response is the so-called **MBoolResponse** class. This class contains in addition to the Boolean value also a list of string in order to provide information in case of errors.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

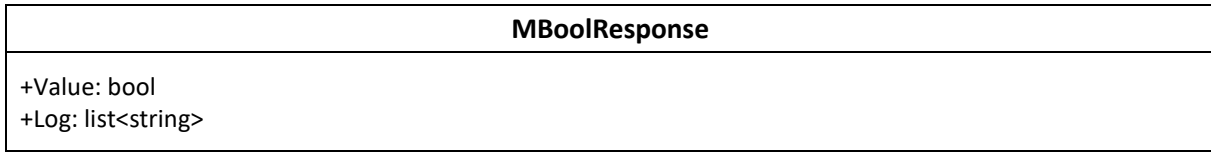


Figure 39 UML class diagram of the MBoolResponse class. The format is used to provide further (debug) information in addition to a Boolean parameter.

Detailed description of available parameters:

Parameter Name	Required	Description
Value	x	A Boolean value which indicates whether the operation was successful or not
Log		An optional list of log strings. This should be used if errors occurred during the processing. This information is helpful for debugging and error handling.

The **MParameter** is an important format for the MMUDescription. The class defines the properties of a specific parameter that can be used by the MMU consumer.

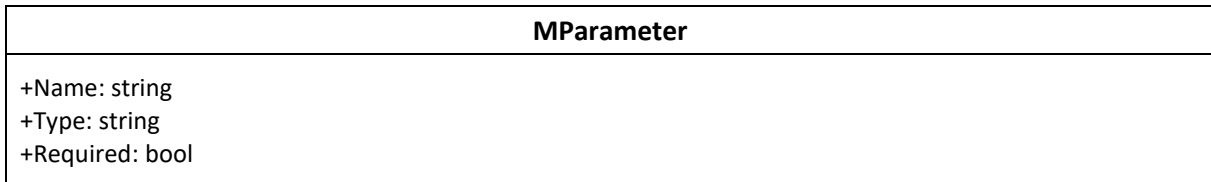
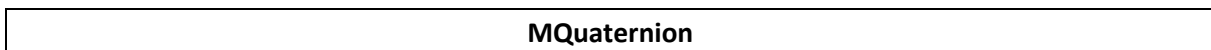


Figure 40 Class diagram of the MParameter.

Detailed description of available parameters:

Parameter Name	Required	Description
Name	x	Unique name of the parameter
Type	x	The type of the parameter (e.g. bool, string, double or user specific type)
Required	x	Specifies whether the parameter must be set to execute the MMU or the parameter is just optionally.

To represent quaternion based rotations in the MMI framework the **MQuaternion** format is utilized.



```

+X: double
+Y: double
+Z: double
+W: double

```

Figure 41 Class diagram of the MQuaternion.

Detailed description of available parameters:

Parameter Name	Required	Description
X	x	The X component of the quaternion.
Y	x	The Y component of the quaternion.
Z	x	The Z component of the quaternion.
W	x	The W component of the quaternion.

For representing a three-dimensional vector with Cartesian coordinates, the **MVector3** class is provided.

MVector3
<pre> +X: double +Y: double +Z: double </pre>



Figure 42 Class diagram of the MVector3.

Detailed description of available parameters:

Parameter Name	Required	Description
X	x	The Cartesian X coordinate.
Y	x	The Cartesian Y coordinate.
Z	x	The Cartesian Z coordinate.

The **MDrawingCall** class is used to exchange information regarding the drawing of geometric elements such as lines. Mainly utilized for debugging.

MDrawingCall

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Type: MDrawingCallType
 Data: list<double>
 Properties: map<string,string>

Figure 43 UML class diagram of the MDrawingCall class.

Detailed description of available parameters:

Parameter Name	Required	Description
Type	x	The specific type of the drawing (see MDrawingCallType enum).
Data	x	The transmitted data (e.g. list of coordinates). The data representations depend on the used MDrawingCallType.
Properties		Additional properties that can be specified.

For the MMI framework, a base set of different drawing types (**MDrawingCallType**) is defined that are listed below.

MDrawingCallType (enum)
DrawLine2D DrawLine3D DrawPoint2D DrawPoint3D

Figure 44 Class diagram of the MDrawingCallType enum, which contains all possible drawing operations in the framework.

Since the overall framework takes intense usage of web-based communication, IP addresses are frequently exchanged. To allow a platform-independent exchange of this information in a standardized way, the **MIPAddress** class is provided.

MIPAddress
+Address: string +Port: int

Figure 45 UML class diagram of the MIPAddress class. The class is utilized to store a network address containing the ip and port.

Detailed description of available parameters:

Parameter Name	Required	Description
Address	x	The address url.
Port	x	The specified port.

3.2.5 Avatar Representation (MAvatarPosture, MAvatarPostureValues)

The focus of the MMI framework is the generation of motions and postures; therewith a format to exchange posture information is essential for the framework. We propose a novel format specifically addressing the requirements of the MMI framework. The bone hierarchy, joint placement and zero-posture of the intermediate representation is pre-defined and constant. However, the dimensions and body parameters of the target avatar can change between different simulations. The definition of the intermediate hierarchy can be found below. The so-called **MAvatarDescription** struct contains this static information, which have to be exchanged during simulation initialization.

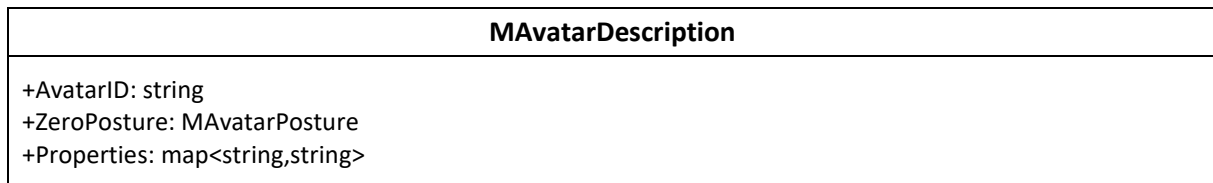


Figure 46 UML class diagram of the MAvatarDescription. The format serves as reference for describing a specific Avatar. In particular, the hierarchy, the id and further properties are defined in here.

Detailed description of available parameters:

Parameter Name	Required	Description
AvatarID	x	The unique id of the corresponding avatar.
ZeroPosture	x	The zero posture describes the specific hierarchy of the target avatar. The list consists of multiple Joints (MJoint) which are described below.
Properties		Additional semantic information which can be set (e.g. weight of body parts).

To describe a full posture the so-called **MAvatarPosture** class can be utilized. The struct contains a unique id, the hierarchy, rotations and positions of all joints. The format can be used for transferring a full posture. Moreover, it is planned to provide several utilizations and helper functions for the **MAvatarPosture** class. For instance, the global positions and rotations can be easily determined based on the class. Moreover, coordinate system specific conversions.

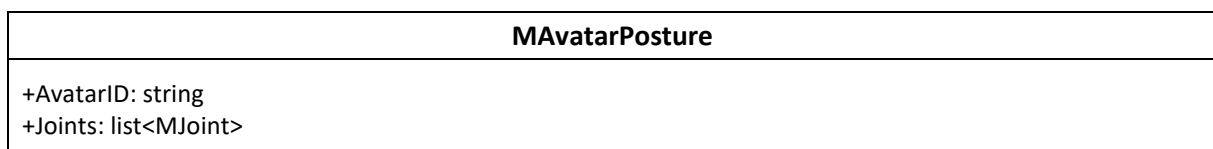


Figure 47 UML class diagram of the MAvatarPosture. The class describes a full posture of a human containing the hierarchy represented as list of MJoint.

Detailed description of available parameters:

Parameter Name	Required	Description
----------------	----------	-------------

AvatarID	x	The unique ID of the avatar.
Joints	x	The joint hierarchy describing the posture.

The **MAvatarPosture** describes the specific hierarchy of the target avatar. The list consists of multiple Joints (**MJoint**) which are described below.

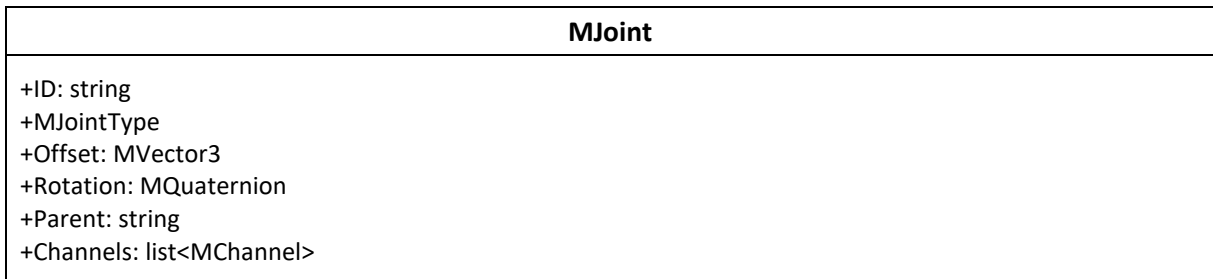


Figure 48 UML class diagram of the MJoint class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	Unique id for the joint.
Offset	x	The position data represented in Cartesian coordinates (analogously to bvh offset).
Rotation	x	The data for rotation represented as quaternion.
Parent		The id of the parent (if available).
Channels	x	Channel information about the representation of the position/rotation (similar to bvh) which is necessary to interpret the transmitted posture values for each frame.

To represent the information of the transferred rotation/position data and its order, the so-called **MChannel** enum is utilized.

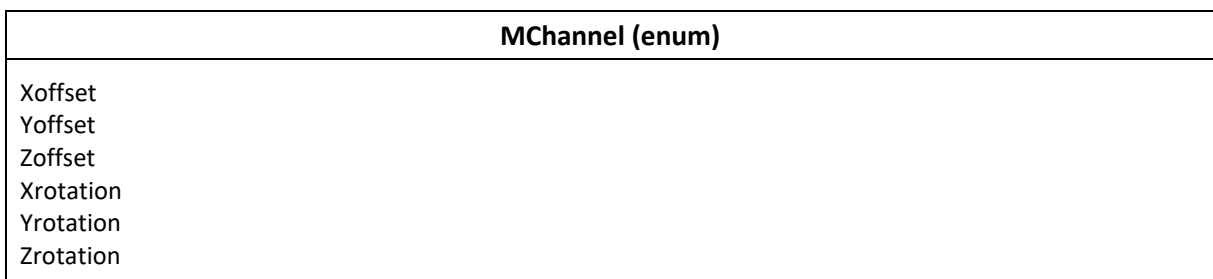


Figure 49 UML class diagram of the MChannel enum. Analogously to the bvh format, the enum describes the specific translation/rotation channel used for transmitting the joint rotation values.

Detailed description of available parameters:

Parameter Name	Description
Xoffset	See bvh
Yoffset	See bvh
Zoffset	See bvh
Xrotation	See bvh
Yrotation	See bvh
Zrotation	See bvh

We assume a right-handed coordinate system with +z as the global up axis and +x as the global forward axis. The joint location is defined in the parent joints coordinate system. The bone orientation is explicitly defined by the offset rotation of the joint. We assume the direction of the bone to be aligned with z-axis of the local coordinate system (see figure 49). Using the channels, rotations around these local axes can be unlocked. Hence, any arbitrary number of channels is valid. In case of the knee-joint, for example, the Channels would be set to [Xrotation, Zrotation], where Xrotation defines the rotation of the knee itself and Zrotation the twist of the calf bone.

The global transform matrix M_{joint} of a single joint can be computed with

With M_{parent} being the parents global transform matrix, O_{joint} the offset and RO_{joint} the rotational offset of the joint. MA_{joint} then describes the animation transform matrix built by subsequently applying the animation rotations and animation offsets in order of the defined channels.

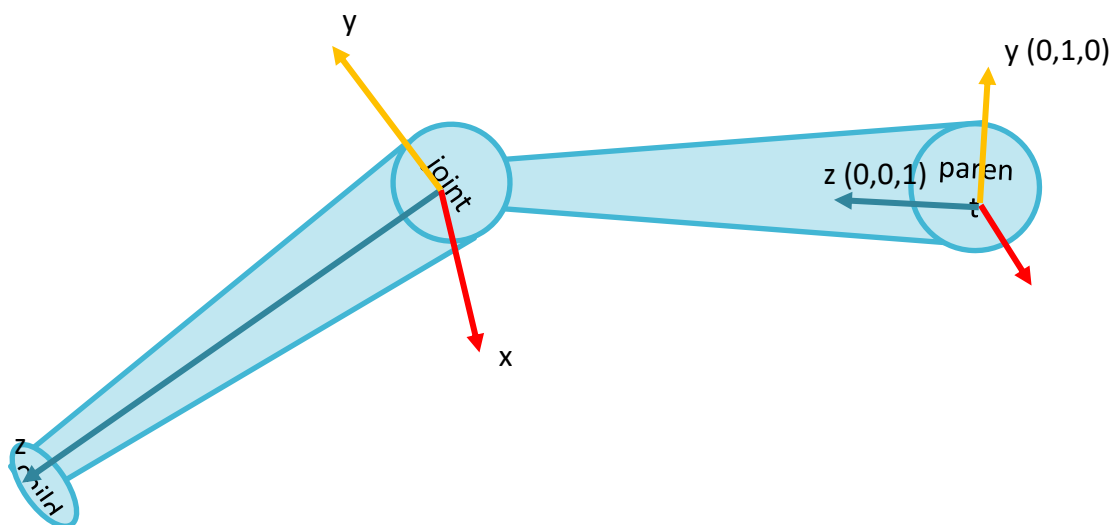


Figure 50 Overview of the coordinate system alignment of the joint chain, as used in the MOSIM project.

The information above is only exchanged during simulation initiation. During the simulation itself, MMUs, Co-Simulator and Target Engine can reduce the information flow to only exchange joint rotation information using **MAvatarPostureValues**. MMUs can further reduce the network traffic by announcing to only transfer a limited amount of data. A grasping MMU, for example, might only want to share rotation information regarding the hands, but not the rest of the body. The joint hierarchy and channel information define the semantics of PostureData. In the appendix (6.2), an example listing is available, which defines the full body hierarchy as well as the joint hierarchy inside the individual hands. They were separated for visual purposes in this document. As a zero posture, a T-posture with parallel legs and feet flat on the ground is expected. The specific joint offsets are depending on the proportions and size of the target avatar.

The **MAvatarPostureValues** are a compact representation of a posture given the assumption that the hierarchy is already known. It is noteworthy that the **MAvatarPostureValues** class should be used as often as possible (and preferred over **MAvatarPosture**) in order to reduce traffic and latency during the communication.

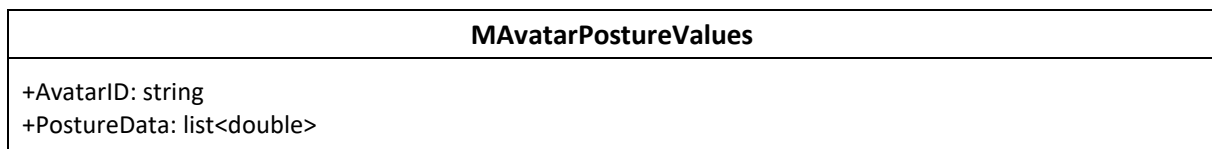
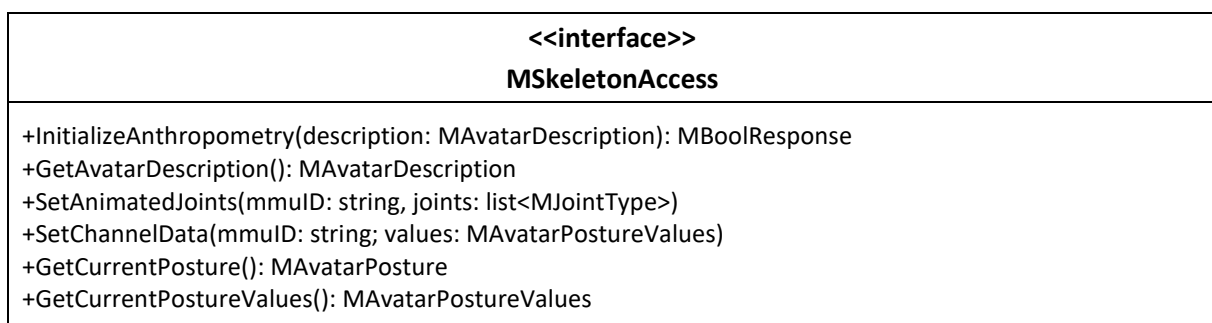


Figure 51 UML class diagram of the MAvatarPostureValues class. The class is used to store joint transformation data.

Detailed description of available parameters:

Parameter Name	Required	Description
AvatarID	x	Unique ID that references the avatar.
PostureData	x	A list representation of the transmitted posture data. The corresponding hierarchy is described as MAvatarPosture (MAvatarDescription) with the identical AvatarID.

The intermediate skeleton as well as skeleton helper functions are managed by the intermediate skeleton service (**MSkeletonAccess**).



```

+GetCurrentJointPositions(): map<MJointType,MVector3>
+GetAvatarRootPosition(): MVector3
+GetAvatarRootRotation(): MQuaternion
+GetGlobalJointPosition(jointType: MJointType): MVector3
+GetGlobalJointRotation(jointType: MJointType): MQuaternion

```

Figure 52 Class diagram of the MSkeletonAccess interface. The interface provides a default set of helper functions to simplify the utilization of the avatar.

Detailed description of available parameters/functions:

Function/ Parameter Name	Description
InitializeAnthropometry	Initializes the internal representation based on the given MAvatarDescription.
GetAvatarDescription	Returns the avatar description (if available).
SetAnimatedJoints	Sets the animated joints.
SetChannelData	Sets the current posture (channel data) based on the given MAvatarPostureValues .
GetCurrentPosture	Returns a MAvatarPosture that describes the current posture (hierarchy + joint values) of the avatar.
GetCurrentPostureValues	Returns MAvatarPostureValues describing the current posture (values) of the avatar.
GetCurrentJointPositions	Returns the global positions of all joints.
GetAvatarRootPosition	Returns the root position of the avatar.
GetAvatarRootRotation	Returns the root rotation of the avatar as quaternion.
GetGlobalJointPosition	Returns the global joint position for the specific joint type.
GetGlobalJointRotation	Returns the global joint rotation for the specific joint type.

To simplify utilization an enum for describing the available joint types (**MJointType**) is available. The enum comprises in total 61 different bone types.

MJointType (enum)
LeftBall
LeftAnkle



MOSIM

End-to-end Digital Integration based on Modular
Simulation of Natural Human Motions
ITEA 3, 17028



Project Coordinator: Thomas Bär, Daimler AG

LeftKnee
LeftHip
RightBall
RightAnkle
RightKnee
RightHip
PelvisCentre
S1L5Joint
T12L12Joint
T1T2Joint
C4C5Joint
HeadJoint
MidEye
LeftShoulder
LeftElbow
LeftWrist
RightShoulder
RightElbow
RightWrist
LeftThumbMidcarpalJoint
LeftThumbMetacarpophalangealJoint
LeftThumbCarpalInterphalangealJoint
LeftIndexMidCarpalJoint
LeftIndexMetacarpophalangealJoint
LeftIndexCarpalProximalInterphalangealJoint
LeftIndexCarpalDistalInterphalangealJoint
LeftMiddleMidCarpalJoint
LeftMiddleMetacarpophalangealJoint
LeftMiddleCarpalProximalInterphalangealJoint
LeftMiddleCarpalDistalInterphalangealJoint
LeftRingMidCarpalJoint
LeftRingMetacarpophalangealJoint
LeftRingCarpalProximalInterphalangealJoint

```

LeftRingCarpalDistalInterphalangealJoint
LeftLittleMidCarpalJoint
LeftLittleMetacarpophalangealJoint
LeftLittleCarpalProximalInterphalangealJoint
LeftLittleCarpalDistalInterphalangealJoint
RightThumbMidcarpalJoint
RightThumbMetacarpophalangealJoint
RightThumbCarpalInterphalangealJoint
RightIndexMidCarpalJoint
RightIndexMetacarpophalangealJoint
RightIndexCarpalProximalInterphalangealJoint
RightIndexCarpalDistalInterphalangealJoint
RightMiddleMidCarpalJoint
RightMiddleMetacarpophalangealJoint
RightMiddleCarpalProximalInterphalangealJoint
RightMiddleCarpalDistalInterphalangealJoint
RightRingMidCarpalJoint
RightRingMetacarpophalangealJoint
RightRingCarpalProximalInterphalangealJoint
RightRingCarpalDistalInterphalangealJoint
RightLittleMidCarpalJoint
RightLittleMetacarpophalangealJoint
RightLittleCarpalProximalInterphalangealJoint
RightLittleCarpalDistalInterphalangealJoint
    
```

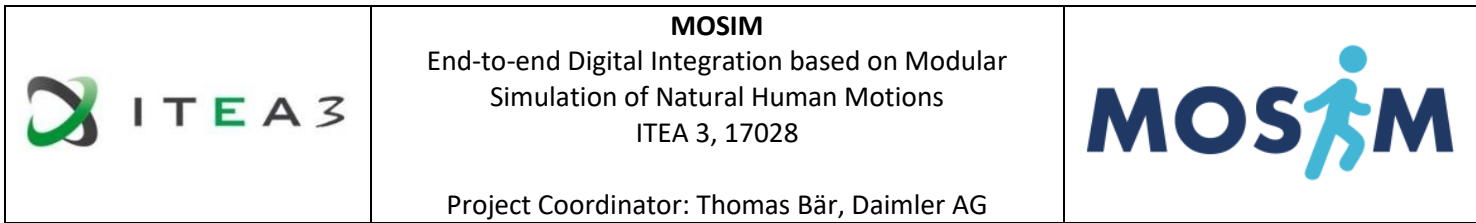
Figure 53 Overview of the MJointType enum, which describes the standardized joint types of the intermediate skeleton.

3.2.6 Scene

The scene serves as central point for storing information related to the simulation environment. In particular, the scene provides access to individual scene objects and avatars. In the following an overview of the essential formats of the scene is given.

The **MSceneObject** builds the basic block for representing a scene object within the MMI framework. Each **MSceneObject** has a unique ID, a required name and a necessary transform. Moreover, optionally a collider, mesh, physic properties and meta-information (properties) can be defined.

MSceneObject



```

+ID: string
+Name: string
+Transform: MTransform
+Collider: MCollider
+Mesh: MMesh
+PhysicsProperties: MPhysicsProperties
+Properties: map<string,string>

```

Figure 54 UML class diagram of an MSceneObject.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	Unique id of the scene object.
Name	x	Name for the scene object.
Transform	x	The mandatory transform of the scene object.
Collider		The collider of the scene object (optional).
Mesh		The mesh of the scene object (optional).
PhysicsProperties		The physical properties of the scene object.
Properties		Optional properties/meta information of the scene object.

The **MTransform** class is relevant for describing the absolute location and hierarchy of a specific object. The overall concept is strongly inspired by the Computer Graphics domain and is widely used in Gaming Engines such as Unity or Unreal.

```

MTransform
+ID: string
+Position: MVector3
+Rotation: MQuaternion
+Parent: string

```

Figure 55 UML class diagram of the MTransform class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The id of the transform (linked to the MSceneObject)
Position	x	The specified position (Cartesian) represented as MVector3 to simplify the usage.

Rotation	x	The specified rotation (Quaternion), represented as MQuaternion to simplify the usage.
Parent		The id of the parent transform/object.

For representing colliders in the proposed framework, the so-called MCollider class is proposed. Since Apache Thrift does not provide class inheritance for structs, the different collider types are explicitly represented within the **MCollider** class. The type of the assigned collider can be directly evaluated based on the **MColliderType** enum.

MCollider
+ID: string +Type: MColliderType +BoxColliderProperties: MBoxColliderProperties +SphereColliderProperties: MSphereColliderProperties +ConeColliderProperties: MConeColliderProperties +CapsuleColliderProperties: MCapsuleColliderProperties +CylinderColliderProperties: MCylinderColliderProperties +MeshColliderProperties: MMeshColliderProperties +PositionOffset: MVector3 +RotationOffset: MQuaternion +Properties: map<string,string> +Colliders: list<MCollider>

Figure 56 UML class diagram of the MCollider class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The id of the collider (linked to the MSceneObject)
Type	x	The type of the collider (defined through the MColliderType enum)
BoxColliderProperties		Specific properties required for a box collider.
SphereColliderProperties		Specific properties required for a sphere collider.
ConeColliderProperties		Specific properties required for a cone collider.
CapsuleColliderProperties		Specific properties required for a capsule collider.
CylinderColliderProperties		Specific properties required for a cylinder collider.
MeshColliderProperties		Specific properties required for a mesh collider.

PositionOffset		The translational offset of the collider to the reference MSceneObject/ MTransform.
RotationOffset		The rotational offset of the collider to the reference MSceneObject/ MTransform.
Properties		Optional properties, which can be set.
Colliders		An optional list for representing a collider hierarchy.

The **MColliderType** enum contains all possible collider types in the framework.

MColliderType (enum)
BoxCollider SphereCollider CapsuleCollider ConeCollider CylinderCollider MeshCollider Custom

Figure 57 UML class diagram of the MColliderType enum. The enum contains all available collider types of the proposed framework.

Each collider has different properties describing its characteristics. To represent these characteristics in the framework for each available **MColliderType**, a respective ColliderProperties class is provided. An overview of the available Collider classes is given below.

MBoxColliderProperties
+Size: MVector3

Figure 58 UML class diagram of the MBoxColliderProperties used to describe the characteristics of a box collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Size	x	The dimensions of the box collider [m].

MSphereColliderProperties
+Radius: double

Figure 59 UML class diagram of the MSphereColliderProperties used to describe the characteristics of a sphere collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Radius	x	The radius of the sphere collider [m].

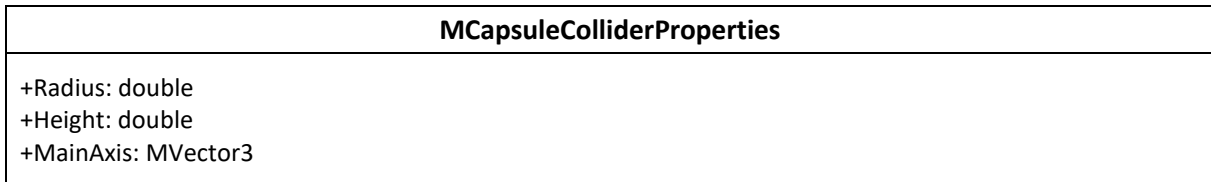


Figure 60 Class diagram of the MCapsuleColliderProperties used to describe the characteristics of a capsule collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Radius	x	The radius of the capsule collider [m].
Height	x	The height of the capsule collider [m].
MainAxis		The main axis of the capsule collider.

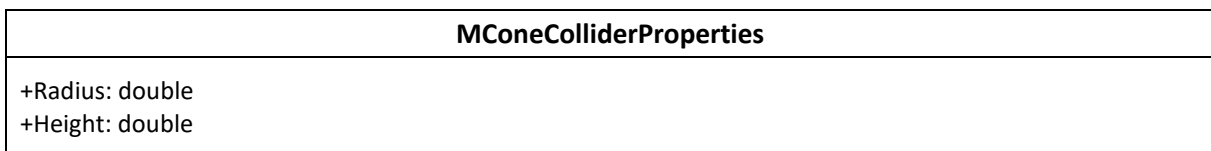


Figure 61 Class diagram of the MConeColliderProperties used to describe the characteristics of a cone collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Radius	x	The radius of the cone collider [m].
Height	x	The height of the cone collider [m].

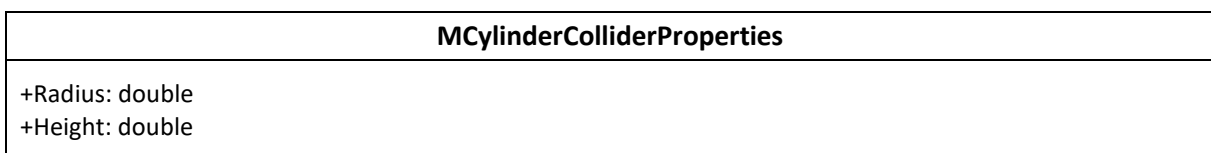


Figure 62 Class diagram of the MCylinderCollider Properties used to describe the characteristics of a cylinder collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Radius	x	The radius of the cylinder collider [m].
Height	x	The height of the cylinder collider [m].

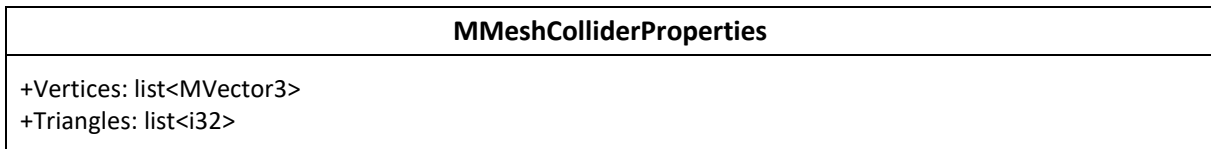


Figure 63 Class diagram of the MMeshColliderProperties used to describe the characteristics of a mesh collider.

Detailed description of available parameters:

Parameter Name	Required	Description
Vertices	x	A list of all vertices of the collider.
Triangles	x	All triangle indices of the collider.

In addition to describing a collider, it is oftentimes required to describe a mesh that is separated from the collider. To describe a mesh in the MMI framework the so-called **MMesh** class is proposed. Below the class diagram, as well as a detailed description of the available parameters is provided.

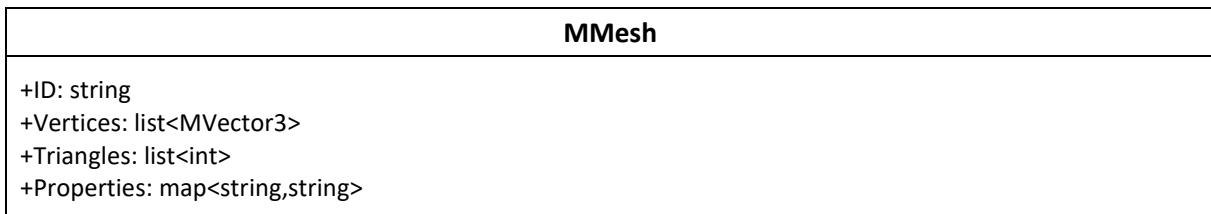


Figure 64 UML class diagram of the MMesh class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The unique ID of the object.
Vertices	x	The vertices of the mesh relative to the MTransform .
Triangles	x	A list of the available triangles of the mesh. Each entry in the list represents the corresponding vertex index.

Properties	Optional properties for the mesh.
-------------------	-----------------------------------

In addition to describing a mesh, it is sometimes required to describe a geometrical component that is separated from the mesh. To describe a geometrical component in the MMI framework the so-called **MGeometry** class is proposed. Below the class diagram, as well as a detailed description of the available parameters is provided.

MGeometry
+ID: string +GeometryType: MGeometryType +SupportPoints: list<list<MVector3>> +Nodes: list<list<double>> +Weights: list<list<double>> +Properties: map<string,string>

Figure 65 UML class diagram of the MGeometry class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The unique ID of the object.
GeometryType	x	The type of the geometry object.
SupportPoints	x	The support points of the geometry. 1- dimensional array for 0- and 1-dimensional objects (point, polyline and curve) and 2-dimensional array for 2-dimensional objects (plane and surface) .
Nodes		Optional nodes in 1- dimensional array for curve and 2-dimensional array for surface (B-Spline representation).
Weights		Optional weights in 1- dimensional array for curve and 2-dimensional array for surface (B-Spline representation).
Properties		Optional properties for the geometry.

For the MMI framework, a base set of different geometry types (**MGeometryType**) is defined that are listed below.

MGeometryType (enum)

```

Point
Polyline
Curve
Plane
Surface, curve SupportPoints (3) (2xn), Weights (2xn), nodes (2xn)

```

Figure 66 Class diagram of the MGeometryType enum.

Each **MSceneObject** contains additional physical properties, such as mass or velocity, which are represented using the **MPhysicsProperties** format outlined below.

```

MPhysicsProperties
+Mass: double
+CenterOfMass: list<double>
+Inertia: list<double>
+Velocity: list<double>
+AngularVelocity: list<double>

```

Figure 67 UML class diagram of the MPhysicsProperties class. The format describes the physical properties of an object.

Detailed description of available parameters:

Parameter Name	Required	Description
Mass	x	The mass of the object.
CenterOfMass	x	The center of mass of the object.
Inertia	x	The present inertia of the object.
Velocity	x	The present velocity of the object.
AngularVelocity	x	The present angular velocity of the object.

In order to do a pathfinding in the scene, it is important to identify the walkable areas. To quantify the walkable area, the so-called **MNavigationMesh** class is provided.



```

MNavigationMesh
+Vertices: list<MVector3>
+Triangles: list<int>
+Properties: map<string,string>

```

Figure 68 UML class diagram of the MNavigationMesh.

Detailed description of available parameters:

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Parameter Name	Required	Description
Vertices	x	All vertices of the navigation mesh
Triangles	x	The triangle indices of the navigation mesh.
Properties		Optional properties for the navigation mesh.

In many cases, it might be of interest for the MMU or target-engine developers to identify all avatars in the scene (e.g. Collision avoidance, multi-avatar interaction). Therefore, a separate format describing a specific Avatar, its properties and location is required. For describing an avatar in the MOSIM framework, the **MAvatar** class is utilized.

MAvatar
+ID: string +Name: string +Description: MAvatarDescription +PostureValues: MAvatarPostureValues +SceneObjects: list<string> +Properties: map<string,string>

Figure 69 UML class diagram of the MAvatar class, which represents an avatar in the framework.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The unique ID of the avatar.
Name	x	The specified name of the avatar.
Description	x	The corresponding MAvatarDescription that defines the skeleton hierarchy.
PostureValues	x	MAvatarPosture Values which describe the current posture of the avatar.
SceneObjects		Optional: Scene objects linked to the MAvatar (e.g. Mesh, collider for the avatar). The scene objects are represented by the unique id.
Properties		Additional properties for the MAvatar .



The scene can be accessed in a uniform way using the **MSceneAccess**. The class and its provided functionality is described in the following:

<<interface>> MSceneAccess
+GetSceneObjects(): list<MSceneObject> +GetSceneObjectByID(ID: string): MSceneObject +GetSceneObjectByName(name: string): MSceneObject +GetSceneObjectsInRange(position: MVector3, distance: double): list<MSceneObject> +GetColliders(): list<MCollider> +GetColliderByID(ID: string): MCollider +GetCollidersInRange(position: MVector3, distance: double): list<MCollider> +GetMeshes(): list<MMesh> +GetMeshByID(ID: string): MMesh +GetTransforms(): list<MTransform> +GetTransformByID(ID: string): MTransform +GetAvatars(): list<MAvatar> +GetAvatarByID(ID: string): MAvatar +GetAvatarByName(name: string): MAvatar, +GetAvatarsInRange(position: MVector3, distance: double): list<MAvatar> +GetSimulationTime(): double +GetChanges(): list<MSceneUpdate> +GetFullScene(): list<MSceneObject> +GetNavigationMesh(): MNavigationMesh

Figure 70 Overview of the MSceneAccess interface that is used to access the scene.

Detailed description of available parameters/functions:

Function Name	Description
GetSceneObjects	Returns a list of all available scene objects.
GetSceneObjectByID	Returns a scene object based on the specified ID.
GetSceneObjectByName	Returns a scene object based on the given name (might be ambiguous -> First result is returned)
GetSceneObjectsInRange	Returns all scene objects within the specified range.
GetColliders	Returns all colliders within the scene.
GetColliderByID	Returns a collider based on the specified id.
GetCollidersInRange	Returns all MCollider within the specified range (intersecting with sphere as defined by the position and radius).
GetMeshes	Returns all meshes within the scene.
GetMeshByID	Returns a mesh based on the given id.
GetTransforms	Returns all transforms within the scene.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

GetTransformByID	Returns a transform based on a given id.
GetAvatars	Returns a list of all available MAvatars .
GetAvatarByID	Returns a MAvatar based on the given id.
GetAvatarByName	Returns a MAvatar based on the given name.
GetAvatarsInRange	Returns all MAvatars within the specified range.
GetSimulationTime	Returns the absolute time of the simulation.
GetChanges	Returns the changes since the last frame.
GetFullScene	Returns the MSceneManipulation to transfer the full scene.

Despite the previously introduced interface, which is designed from a user perspective, the scene in the target engine, or nested co-simulation needs to provide further synchronization and updating functionalities. For this purpose, the **MSynchronizableScene** interface is provided.

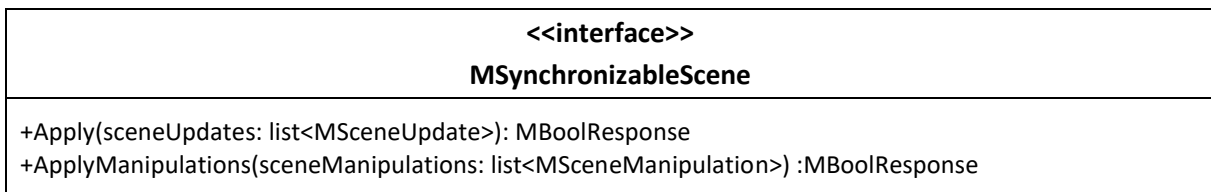


Figure 71 Class diagram of the MSynchronizableScene interface.

Detailed description of available parameters/functions:

Function Name	Description
Apply	Applies transmitted MSceneUpdates to the scene. For instance, this might be utilized for a scene contained in an adapter, which continuously buffers the scene (only deltas are transmitted).
ApplyManipulations	Function to apply manipulation as expressed using the MSceneManipulation class. The function is utilized in the target engine/ by the co-simulation to apply manipulations as intended by MMUs (e.g., move object).

Formats for transferring scene information:

In order to allow an efficient transfer of the overall scene data, further formats optimized with respect to the data sizes are used. Below a detailed overview of these classes is given.

The **MSceneObject** class contains all changed elements within the scene.

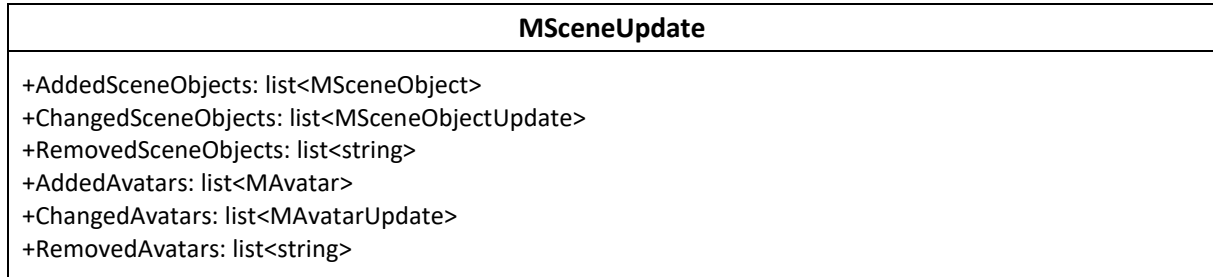


Figure 72 UML class diagram of the MSceneUpdate class. The format is utilized to efficiently represent changes of the scene for transferring them over the network.

Detailed description of available parameters:

Parameter Name	Required	Description
AddedSceneObjects		List of newly added MSceneObjects .
ChangedSceneObjects		List of all scene object changes.
RemovedSceneObjects		List of the ids of all removed scene objects.
AddedAvatars		List of newly added MAvatars .
ChangedAvatars		List of the MAvatarUpdates .
RemovedAvatars		List of the ids of all removed avatars.

For describing and transferring the changes of a specific **MSceneObject**, the **MSceneObjectUpdate** class is used.

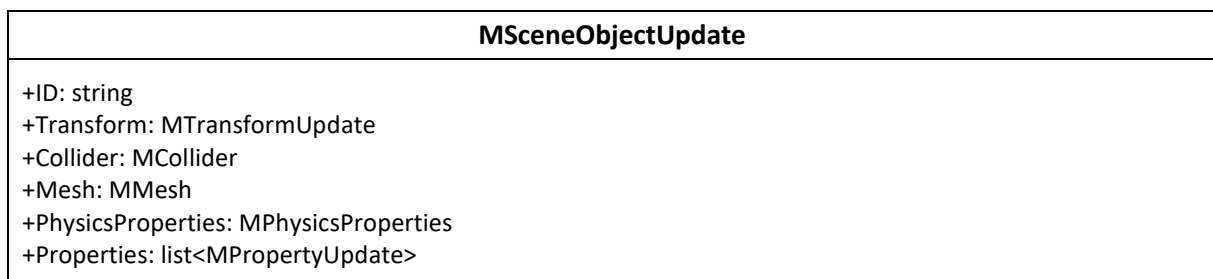


Figure 73 UML class diagram of the MSceneObjectUpdate.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The id of the MSceneObject that should be updated.

Transform		The changed transforms (absolute).
Collider		The collider (if changed).
Mesh		The mesh (if changed).
PhysicsProperties		The changed physics properties.
Properties		A list of changed properties.

To efficiently transfer changed transform values, as frequently required during the simulation, the **MTransformUpdate** class is used.

MTransformUpdate
+Position: list<double> +Rotation: list<double> +Parent: string

Figure 74 UML class diagram of the MTransformUpdate class.

Detailed description of available parameters:

Parameter Name	Required	Description
Position		The changed position.
Rotation		The changed rotation.
Parent		The changed parent.

To describe changes with respect to properties. The **MPropertyUpdate** class is provided, which is outlined below.

MPropertyUpdate
+Key: string +Value:string

Figure 75 UML class diagram of the MPropertyUpdate format.

Parameter Name	Required	Description
Key	x	The key of the property, which is adjusted/added.
Value	x	The value of the property.

The **MAvatarUpdate** class is used to describe all occurred changes regarding a specific MAvatar. In particular, the changes can include changed posture values, properties, the description itself or the involved list of scene object IDs.

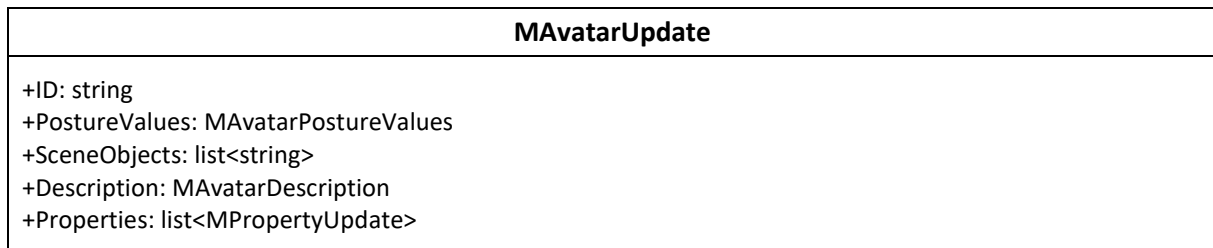


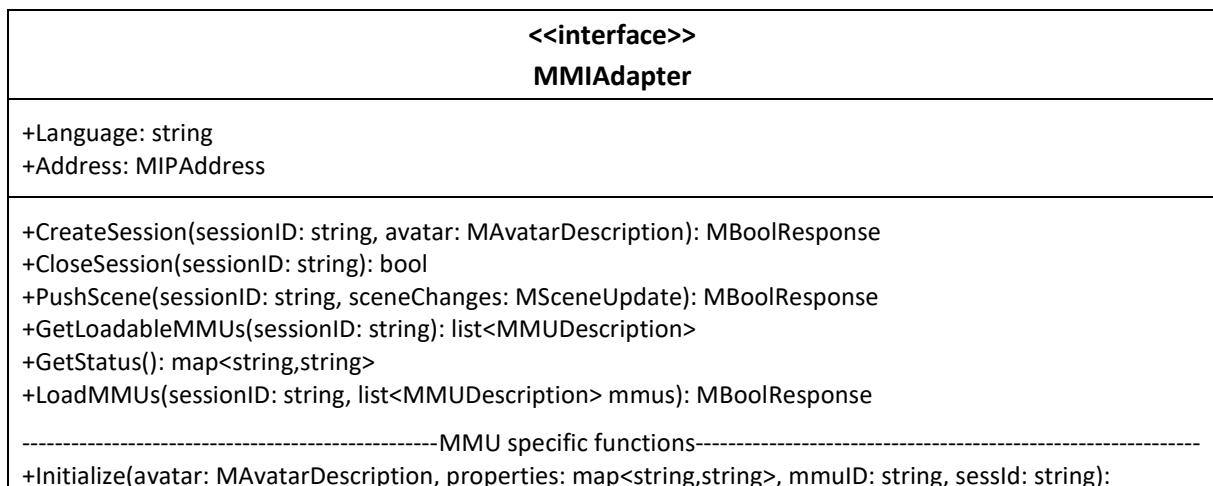
Figure 76 UML class diagram of the MAvatarUpdate class.

Detailed description of available parameters:

Parameter Name	Required	Description
ID	x	The unique ID of the avatar, which should be manipulated.
PostureValues		The changed posture values of the avatar (if defined).
SceneObjects		The changed assignment of scene objects (if defined).
Description		The adjusted description of the avatar (if defined).
Properties		A list of changed properties.

3.2.7 Adapter

Strongly related to the MMUs and the co-simulation, the adapters encapsulate the communication and instantiate the respective MMUs. Therefore, it is important to note, that the interface of the adapter is essential for the data transmission within the MMI framework.



```

MBoolResponse
+AssignInstruction(instr: MInstruction, state: MAvatarState, mmuID: string, sessID: string): MBoolResponse
+DoStep(time: double, state: MAvatarState, mmuID: string, sessID: string): MSimulationResult
+CheckPrerequisites(instr: MInstruction, mmuID: string, sessID: string): MBoolResponse
+GetBoundaryConstraints(instr: MInstruction, mmuID: string, sessID: string): list<MConstraint>
+ExecuteFunction(name: string, parameters: map<string,string>, mmuID: string, sessID: string):
map<string,string>
+Abort(instructionID: string, mmuID: string, sessID: string):MBoolResponse
+Dispose(properties: map<string,string>, mmuID: string, sessID: string):MBoolResponse

```

Figure 77 Interface of the MMIAAdapter. The MMIAAdapter is an essential component, which encapsulate the respective MMUs.

The adapter provides all functionalities, which a single MMU has according to the specified interface. In addition, the adapters also provide functions to manage multiple MMUs and synchronize the scene. In particular, the adapters provide an internal session handling for multiple clients. Moreover, the adapter need to load and instantiate the MMUS during runtime.

Detailed description of available parameters/functions:

Function Name	Description
CreateSession	Method setups a new session. The adapter internally manages the session. The session id is used to access and identify the specific session.
CloseSession	Method closes the current session with the given session id.
PushScene	Method is used to push the deltas of a scene to the adapter.
GetLoadableMMUs	Returns all MMUs found by the adapter, which can be loaded.
GetStatus	Returns the status of the adapter-
LoadMMUs	Loads MMUs based on the specified descriptions.
Initialize	Calls the initialize function of the respective MMU.
AssignInstruction	Calls the AssignInstruction function of the respective MMU.
DoStep	Calls the DoStep function of the respective MMU.
CheckPrerequisites	Calls the CheckPrerequisites function of the respective MMU.
GetBoundaryConstraints	Calls the GetBoundaryConstraints function of the respective MMU.
Abort	Calls the Abort function of the respective MMU.
Dispose	Calls the Dispose function of the respective MMU.

ExecuteFunction	Calls the ExecuteFunction method of the respective MMU.
------------------------	---

Similarly, to the MMUs, the Adapters contain also a description format (**MAdapterDescription**). The class contains useful information about the name and id of the adapter. Furthermore, the address information required for accessing the adapters is contained.

MAdapterDescription
+Name: string +ID: string +Language: string +Addresses: list<MIPAddress> +Properties: map<string,string> +Parameters: list<MParameter>

Figure 78 UML class diagram of the MAdapterDescription.

Detailed description of available parameters:



Parameter Name	Required	Description
Name	x	The name of the adapter.
ID	x	A unique ID of the adapter.
Language	x	The supported programming language of the adapter.
Addresses	x	A list of address under which the adapter is available.
Properties		Optional properties of the adapter.
Parameters		A list of optional parameters of the adapter.

3.2.8 Services

Services are essential to ease the implementation for the MMU developers, co-simulation, behavior execution and target engine users. Within the MMI framework, a set of fundamental services is available. The interfaces of the specific services are listed below.

In analogy to the Adapters, the services also offer a standardized description format.

MServiceDescription

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

```

+Name: string
+ID: string
+Language: string
+Addresses: list<MIPAddress>
+Properties: map<string,string>
+Parameters: list<MParameter>

```

Figure 79 Class diagram of the MServiceDescription format.

Detailed description of available parameters:

Parameter Name	Required	Description
Name	x	The name of the service.
ID	x	A unique ID of the service.
Language	x	The supported programming language of the service.
Addresses	x	A list of address under which the service is available.
Properties		Optional properties of the service.
Parameters		A list of optional parameters of the service.



Each provided service must implement and extend the **MMIServiceBase**. In particular, the base service contains a setup and a generic consume method, as outlined below.

<<interface>> MMIServiceBase
+Setup(description: MAvatarDescription, properties: map<string,string>): MBoolResponse +Consume(properties: map<string,string>): map<string,string> +GetStatus(): map<string,string> +GetDescription(): MServiceDescription

Figure 80 Class diagram of the MMIServiceBase interface. The interface must be implemented by all services in the MOSIM framework.

Detailed description of available parameters/functions:

Function Name	Description
Setup	Basic method to setup the service. This function can be used to reduce the network traffic. For instance, instead of transferring the full hierarchy, only the posture values can be transmitted if being initialized in before.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Consume	Function to consume a service without needing the explicit interface. This can be utilized if new services are added to the framework which signature is not known yet.
GetStatus	Returns the present status of the service.
GetDescription	Returns the specific MServiceDescription for the service.

The **MRetargetingService** depicts a fundamental service within the proposed MMI framework. In particular, the MMU developers can utilize the service to map between the intermediate and the specific skeleton. Moreover, the target engine developers can further take usage of the proposed service.

<<interface>> MRetargetingService
+SetupRetargeting(interm: MAvatarDescription, specific: MAvatarDescription, id: string): MBoolResponse +RetargetToIntermediate(values: MAvatarPostureValues, id: string): MAvatarPostureValues +RetargetToSpecific(values: MAvatarPostureValues, id: string): MAvatarPostureValues

Figure 81 Overview of the interface of the MRetargetingService.

Detailed description of available parameters/functions:



Function Name	Description
SetupRetargeting	Sets up the retargeting given the different postures.
RetargetToIntermediate	Retargets the posture values to the intermediate skeleton.
RetargetToSpecific	Retargets the posture values from the intermediate skeleton to the specific skeleton.

A further service, which is provided by the MMI framework, is the **MPathPlanningService**. The service allows computing a collision-free path given a start and end-configuration. In particular, the service provides multiple operations modes for computing paths with different dimensionalities and criteria.

<<interface>> MPathPlanningService
+ComputePath(start: list<double>, goal: list<double>, objects: list<MSceneObject>, properties: map<string,string>): list<double>

Figure 82 Overview of the interface of the MPathPlanningService.

Detailed description of available parameters/functions:

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Function Name	Description
ComputePath	Computes a path using the specified parameters and scene objects.

An essential service within the framework is the **MCollisionDetectionService**. The service allows detecting collisions between specified colliders, as frequently required for digital human simulations.

MCollisionDetectionService
+ComputePenetration(a: MCollider, b: MCollider, aPos: MVector3, bPos: MVector3, aRot: MQuaternion, bRot: MQuaternion): MVector3
+CausesCollision(a: MCollider, b: MCollider, aPos: MVector3, bPos: MVector3, aRot: MQuaternion, bRot: MQuaternion): MBoolResponse

Figure 83 Overview of the interface of the MCollisionDetectionService.

Detailed description of available parameters/functions:

Function Name	Description
ComputePenetration	Computes the penetration between two colliders given their hypothetical position/rotation.
CausesCollision	Indicates whether the two specified colliders cause a collision given the position/rotation.

Since inverse kinematics is a fundamental aspect of human motion synthesis and used throughout different technologies, the framework also provides a service for computation of inverse kinematics.

MInverseKinematicsService
+ComputeIK(posture: MAvatarPosture, properties: list<MIKProperty>):MAvatarPosture
+ComputeIK(postureValues: MAvatarPostureValues, properties: list<MIKProperty>): MAvatarPostureValues

Figure 84 Interface description of the MInverseKinematicsService.

Detailed description of available parameters/functions:

Function Name	Description
ComputeIK	The method computes a novel posture based on the given MIKProperties . In particular, the posture values of the resulting posture are returned.

The IK service comes with a set of additional structures that simplify the utilization of the service. The provided formats are presented below:

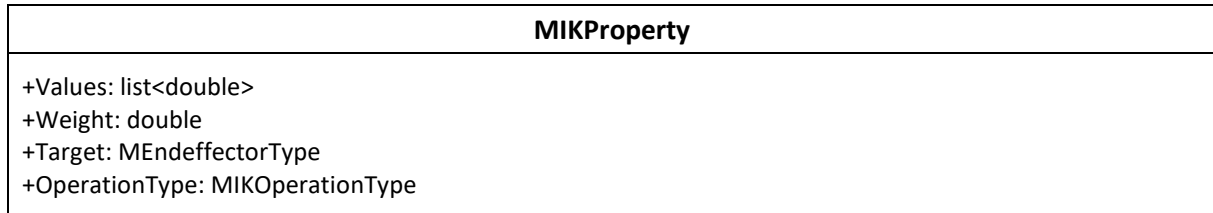


Figure 85 UML class diagram of the MIKProperty class.

Parameter	Required	Description
Values	x	The values for the specified MIKOperationType (e.g. positions, rotations).
Weight	x	The weight of the property ([0; 1]).
Target	x	The MEndeffectorType, which should be adjusted.
OperationType	x	The specific operation that should be carried out using the IK.

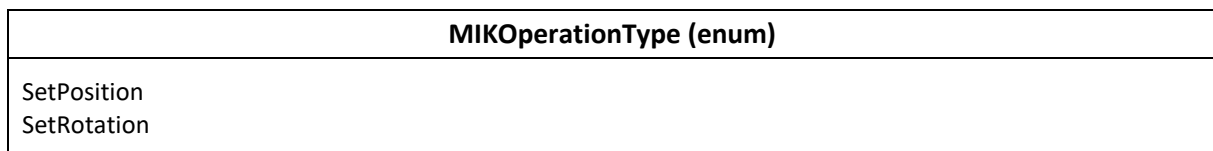


Figure 86 Class diagram of the MIKOperationType enum.

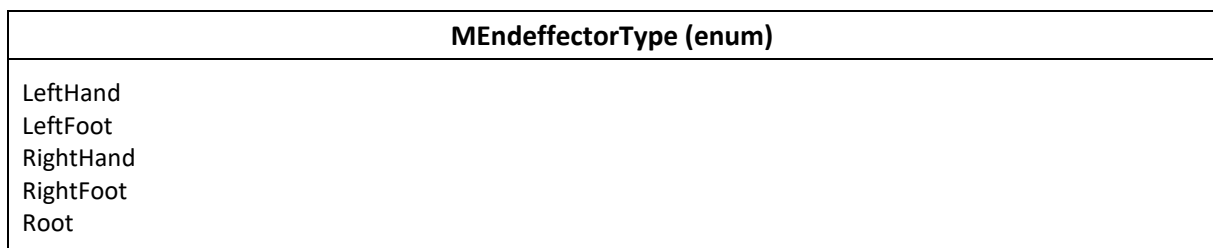


Figure 87 Class diagram of the MEndeffectorType enum.

To blend between consecutive posture for motion synthesis or transition modeling, the so-called **MBlendingService** is provided in the framework.

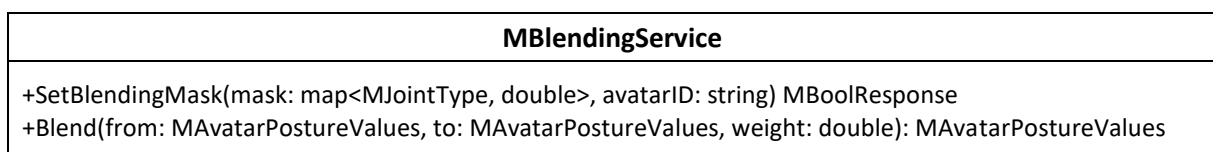




Figure 88 Interface of the MBlendingService, which provides functionality for motion blending.

Detailed description of available parameters/functions:

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

Function Name	Description
SetBlendingMask	Method sets the blending mask for the given avatar id.
Blend	Method performs a blend from the start to the target posture using the specified blend weight.

To access all services, the **MServiceAccess** interface is provided. It is accessible from the MMUs and co-simulation. It contains all defined interfaces of the previously described services. Since Apache Thrift does not allow using services as class members, the specific **MServiceAccess** must be manually provided for each Adapter implementation.

<<interface>> MServiceAccess
+InverseKinematics: MInverseKinematicsService +Retargeting: MRetargetingService +PathPlanning: MPathPlanningService +CollisionDetection: MCollisionDetectionService +MotionBlending: MMotionBlendingService

Figure 89 Overview of the interface of the MServiceAccess. The interface serves as a basic accessing functionality for all provided services in the framework.

Detailed description of available parameters/functions:

Function Name	Description
InverseKinematics	Access to the InverseKinematics service
Retargeting	Access to the retargeting service
PathPlanning	Access to the path planning service
CollisionDetection	Access to the collision detection service
MotionBlending	Access to the motion blending service

3.2.9 Launcher (MMIRegisterService)

The launcher is the main component that starts the overall MMI environment. In particular, the component can optionally start all services and the available adapters using the **MExecutableDescription** file. Moreover, the component serves as central registry and provides information regarding the available adapters, services and MMUs. Especially, the target engine uses the Launcher to access the available Motion Model Units. It can be therefore considered as logical communication layer.

<<interface>> MMIRegisterService
<pre> +GetRegisteredAdapters(): list<MAdapterDescription> +GetRegisteredServices(): list<MServiceDescription> +GetAvailableMMUs(): map<MMUDescription,list<MIPAddress>> +RegisterAdapter(description: MAdapterDescription) +UnregisterAdapter(description: MAdapterDescription) +RegisterService(description: MServiceDescription) +UnregisterService(description: MServiceDescription) +CreateSessionID(): string </pre>

Figure 90 Interface of the MMIRegisterService.

Detailed description of available parameters/functions:

Function Name	Description
GetRegisteredAdapters	Returns all registered Adapters in the current MMI environment.
GetRegisteredServices	Returns all registered Services in the current MMI environment.
GetAvailableMMUs	Returns all available MMUs in the current MMI environment with the respective addresses at which the MMU is accessible.
RegisterAdapter	Allows registering an adapter in the current environment.
UnregisterAdapter	Allows unregistering an adapter in the current environment.
RegisterService	Allows registering a service in the current environment.
UnregisterService	Allows unregistering an adapter in the current environment.
CreateSessionID	Returns a unique session ID

In general, the proposed launcher is provided with the released version of the framework. The launcher is able to start executable applications such as adapters or services. For this purpose a description file is required.

MExecutableDescription
<pre> +Name: string +ID: string +ExecutableName: string +Dependencies list<string> Dependencies +Properties: map<string,string> </pre>

Figure 91 Overview of the MExecutableDescription class.

Detailed description of available parameters/functions:



MOSIM
End-to-end Digital Integration based on Modular
Simulation of Natural Human Motions
ITEA 3, 17028



Project Coordinator: Thomas Bär, Daimler AG

Parameter	Required	Description
Name	x	The name of the application.
ID	x	Unique id of the application.
ExecutableName	x	The name of the executable file (e.g., adapter.exe or service.bat)
Dependencies		Optional dependencies to start the application.
Properties		Optional properties.

3.3 Workflow

Building upon the described interfaces and formats, the specific workflows for the utilization of the MOSIM MMI framework are presented in the following.

3.3.1 Launcher – The entry point for the overall MMI framework

The launcher can be considered as the entry point of the overall MMI framework. The launcher provides a central service, named **MMIRegisterService** (see 3.2), which offers accessing functionalities of all registered adapters/services and available MMUs. Figure 92 gives an overview of the sequence-diagram of the launcher.

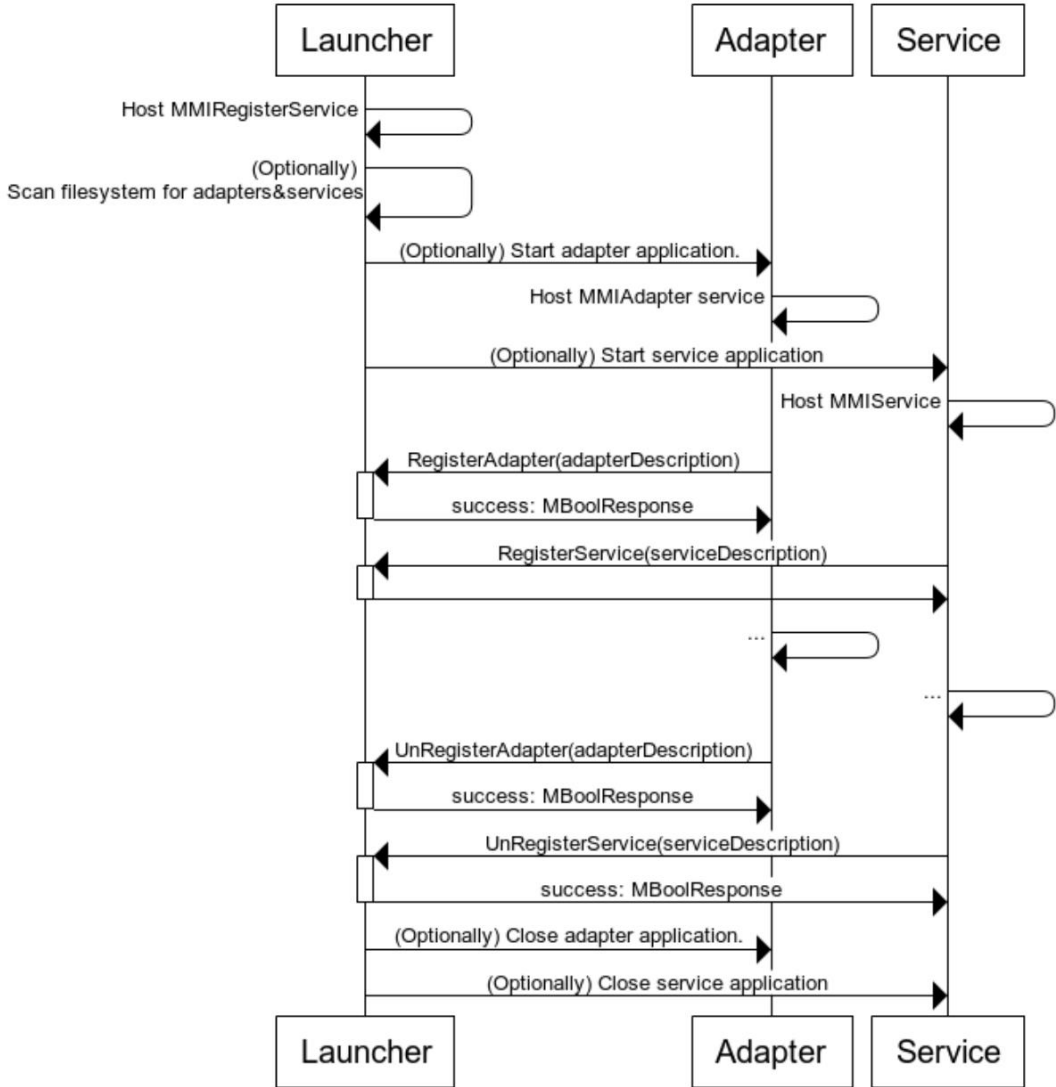




Figure 92 Sequence diagram of the launcher.

The launcher initially starts hosting the **MMIRegisterService** (see 3.2.9) which serves as a central registry for the overall system. Next, the launcher optionally scans the file system for executable adapters and service applications (using the **MExecutableDescription**).

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

automatically starts and manages the respective processes. Each adapter and service registers itself at the central register using the specific **MAdapterDescription** or **MServiceDescription**. Therefore, it is also possible to add external adapters or services, which were not explicitly started and are not managed by the launcher. If an adapter or service is terminated, the unregister function of the **MMIRegisterService** is used together with the respective **MAdapterDescription** or **MServiceDescription**. From this point, the adapter and service are not further listed as available. Moreover, if the launcher internally started applications such as adapters or services, the applications are automatically closed once the launcher is shut down. The launcher component is provided in addition to this document as part of the MOSIM deliverable.

3.3.2 MMU Execution

Despite the description of the workflow of the launcher, which serves as the entry point, it is also crucial to define the workflow of the specific MMUs. In general, the MMU execution can be subdivided into different phases, namely Initialization, Computation and Termination. The sequence diagram below (see Figure 93) illustrates the overall workflow of executing an MMU from an MMU perspective.

First, the MMU is initialized given the **MAvatarDescription** of the intermediate skeleton as well as additional properties. The MMU indicates whether the initialization was successful by returning a **MBoolResponse**. After initialization, the superior instance or consumer might call the **GetBoundaryConstraints** method of the MMU. Therewith, the MMU needs to return boundary constraints, which are relevant to start/end the motion. In particular, these constraints are important for the transition modeling (e.g., start posture). For a data-driven MMU that solely plays back a motion, the returned boundary constraint could be the first posture of the played back motion.

Next, the **CheckPrerequisites** function is called by the consumer/co-simulator. The MMU is responsible for indicating whether the intended instruction can be executed given the present state of the scene.

Within the computation phase, the **AssignInstruction** method depicts a major functionality. In order to compute and express the desired motion, the **AssignInstruction** method is executed given a specified **MInstruction** and the present **MSimulationState**. The method is only executed if the **CheckPrerequisites** function of the MMU returns true. This method might require more time than actually available for real-time computation. For instance, the internal model could be set up, paths or entire motions could be pre-computed. The MMU returns a status whether the assignment was successful (**MBoolResponse**). Being executed for every frame, the **DoStep** method is responsible for the actual computation/generation of the individual frames and postures. Therewith, the function gets the delta time (time to be simulated), as well as the present simulation state (**MSimulationState**) as input. The method returns the generated **MSimulationResult**. The **DoStep** method is executed until the MMU generates an End Event expressed using the **MSimulationResult** or the consumer aborts the instruction using the **Abort** method. Within the **Abort** method, the MMU is responsible to terminate the specified instruction and reset the internal state to the initial state. Furthermore, if the simulation environment/MMU is completely terminated, the **Dispose** method is used.

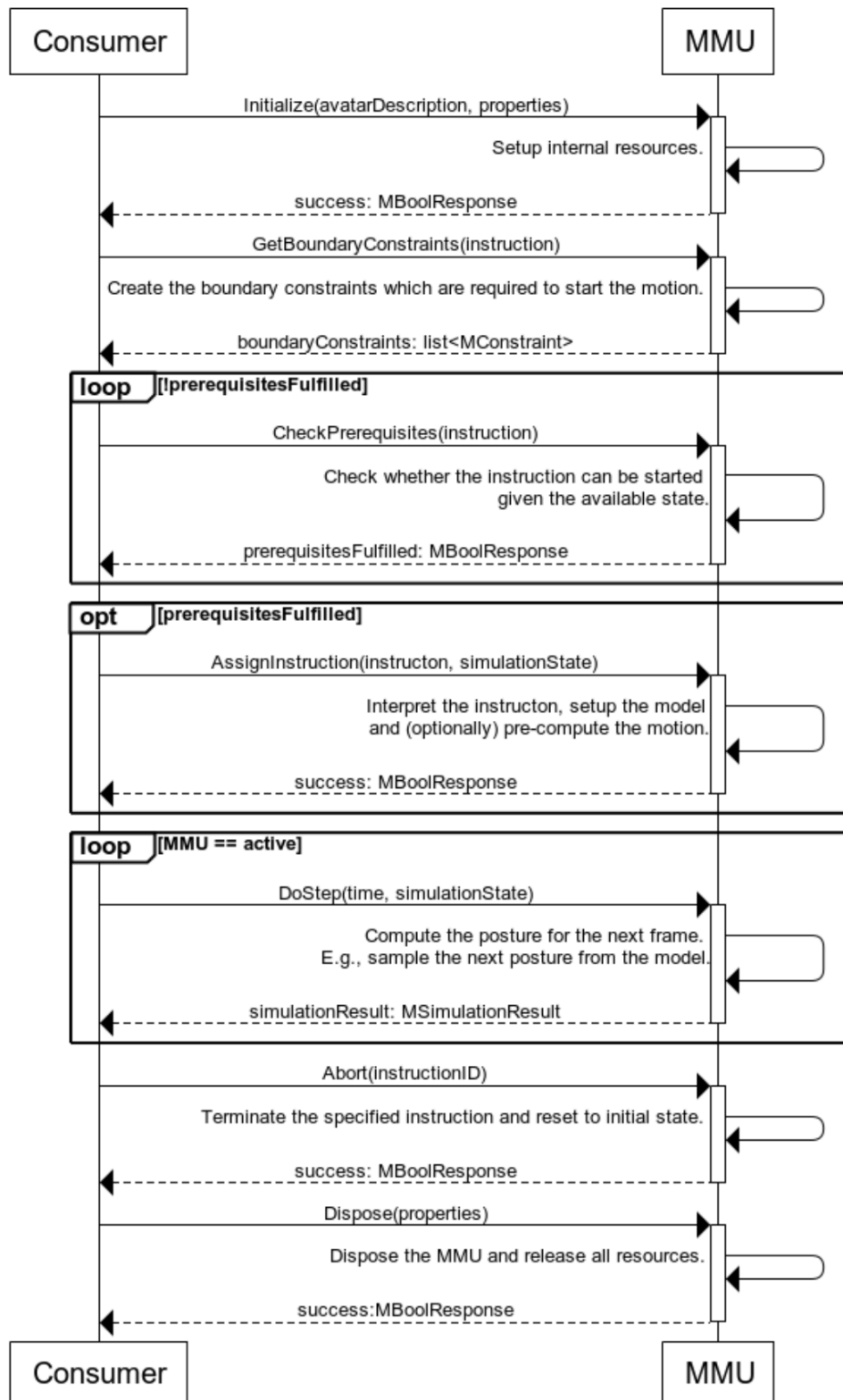




Figure 93 Sequence diagram describing the workflow of the MMU utilization.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

3.3.3 Adapters

The adapter forms an essential aspect of the framework, encapsulating the MMUs in the respective programming language while buffering the scene. Similar to the MMUs, the adapter workflow is subdivided into different phases that are described in the following.

Initialization phase:

In general, the external consumer or launcher starts the adapter application at the beginning. In particular, during startup several arguments must be passed to the adapter. The address of the desired **MMIAdapter** server (-a), the address of the **MMIRegisterService** (-r) and the file path or url of the MMU directory (-m) must be provided as console arguments. An exemplary set of startup arguments is illustrated below:

```
-a 127.0.0.1:8000 -r 127.0.0.1:9009 -m C:\MMUs\
```

After the adapter application has been started, the adapter hosts the respective **MMIAdapter** thrift server to be accessible from external consumers, whereas the specified address parameter (-a) is used as hosting address. Note that the implementation for the **MMIAdapter** must be specifically provided for each programming language. Furthermore, the adapter scans the file system or remote location, as specified using the -m parameter, for compatible MMUs. In particular, the **MMUDescription** files are utilized for analyzing the compatibility. If the **MMUDescription** contains the same programming language and supported dependencies as the adapter, the MMU is considered as compatible. The list of compatible MMUs and resources is internally stored within the adapter. In order to be visible within the MOSIM framework, next, the adapter registers itself at the **MMIRegisterService** using the register address provided as argument.

After the adapter has been successfully registered at the launcher, the adapter is now visible for all clients. From here on the client can create a session at the adapter using a specified session ID. It is noteworthy, that the adapter manages an individual set of MMUs, avatars and scenes for each sessionID. Given the created session, the client can now request the loadable MMUs and select the desired MMUs to be loaded. The adapter internally instantiates the respective MMUs and returns a status flag indicating whether the operation was successful. The instantiating process strongly deviates between the heterogeneous programming languages. For a C# adapter, the instantiation is realized using the Reflection functionalities of the .Net framework and .dll files, whereas for Java .jar files are utilized. Therefore, the instantiation needs to be manually implemented for each programming language.

After the MMUs have been instantiated, the client can initialize the given MMU with the respective mmuID and sessionID. The mmuID is provided within the **MMUDescription** file, read by the adapter. In before, the client pushes the full scene to the adapter using the **PushScene** method.

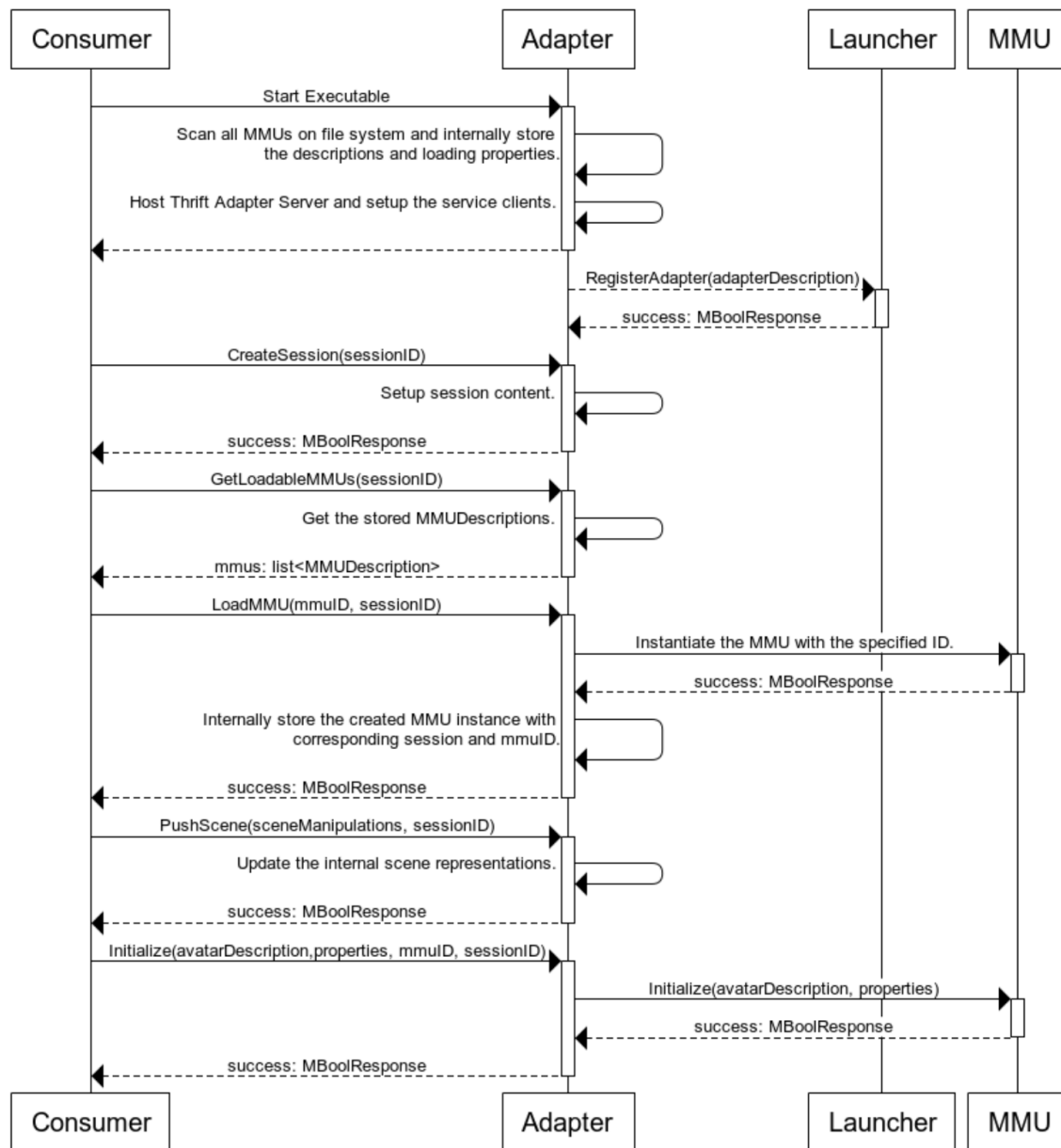


Figure 94 Illustration of the initialization phase of the adapter utilizing a sequence diagram.

Computation phase:

After the initialization of the adapter and the respective MMUs, next the actual motion can be computed within the computation phase. Figure 95 visualizes the corresponding sequence diagram. During the computation phase, the consumer pushes the scene changes to the adapter at the beginning of each frame, ultimately resulting in a synchronized scene on a per frame basis. For this purpose, the **PushScene** method of the adapter is utilized, transmitting the delta scene changes using the **MSceneUpdate** class. Next, similar to the MMU workflow, the prerequisites of the instruction are checked. If the prerequisites are fulfilled the instruction is assigned. Different to the direct accessing of the MMUs, at each call, the sessionID and mmuID must be additionally specified to allow the adapter identifying the respective MMUs. For the actual frame wise computation of the motions, the **DoStep** method of the adapter with the specified mmuID and sessionID. Internally, again, the respective MMU

is called and the result returned to the consumer. The adapter therefore acts as proxy between the consumer and MMU.

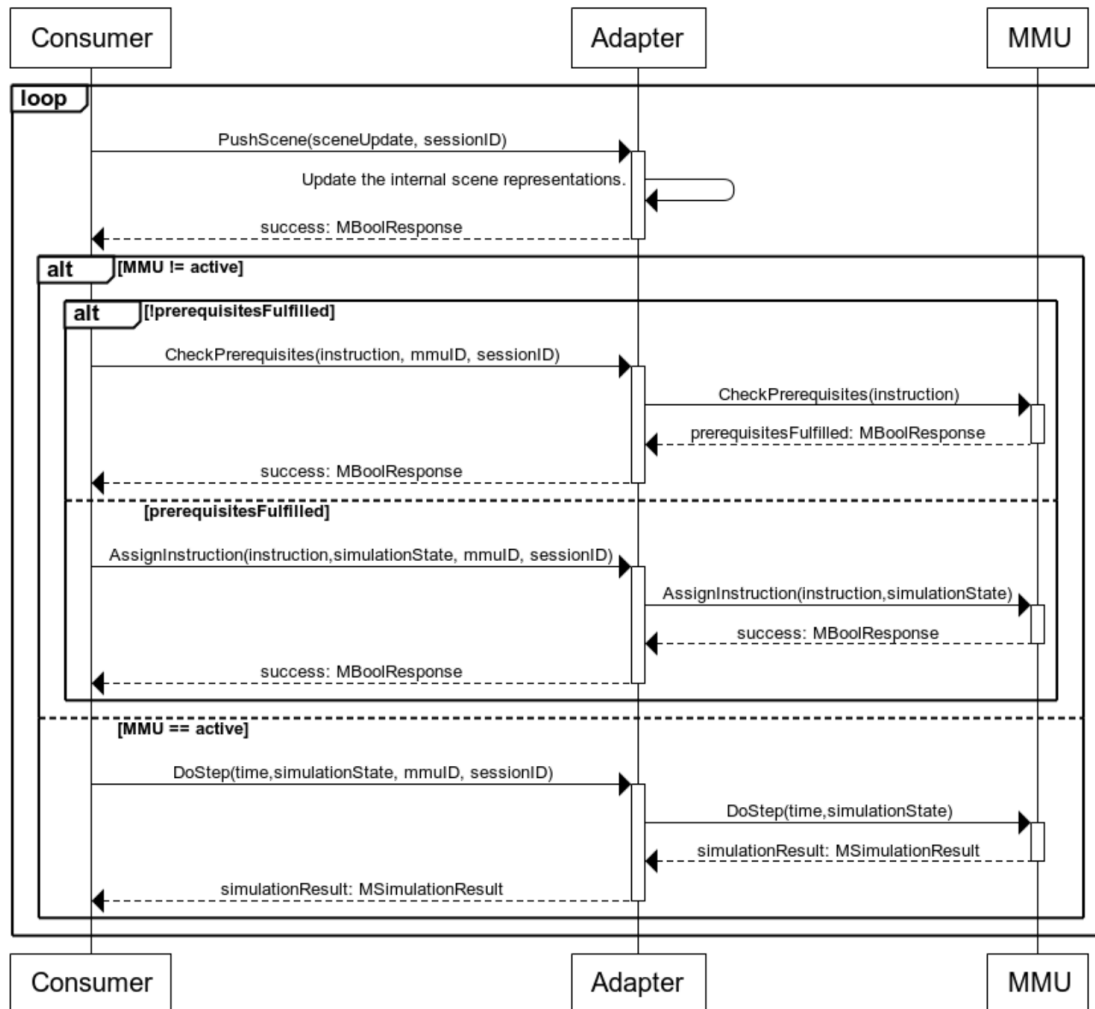


Figure 95 Sequence diagram of the computation phase of the adapter.

Termination phase:

To terminate specific MMUs, the **Dispose** function specifying the mmuID and session ID is used. Moreover, after all MMUs have been disposed, the session can be closed using the **CloseSession** method. Internally, the adapter releases the allocated resources for the specified session. Finally, the consumer must terminate the adapter application (process). If the adapter process is closed, the adapter unregisters at the central Launcher using the **Unregister** method. In this way, the adapter is not any more visible to further consumers in the MOSIM framework environment.

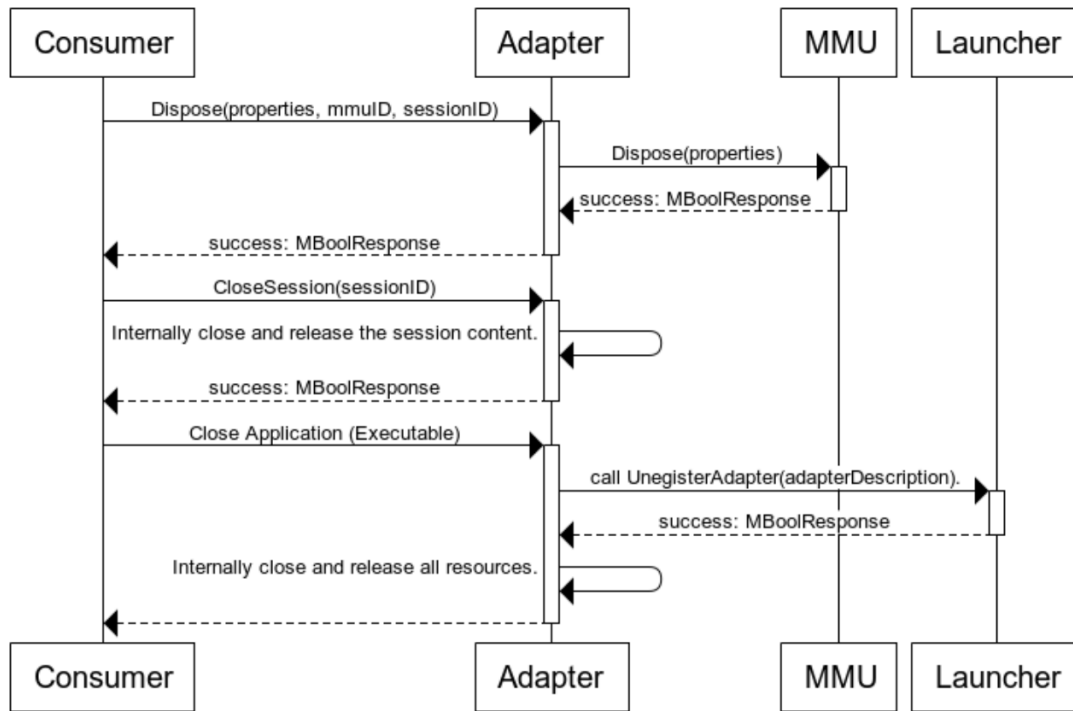


Figure 96 Sequence diagram of the termination phase of the adapter.

3.3.4 Co-Simulation

The co-simulation is generally responsible for executing the MMUs and generating an approved motion based on the underlying simulation approaches. As input, a sequence of instructions, as generated by the behavior execution is provided. The output is a feasible human posture for each frame. In the following, the execution sequence for the co-simulation is illustrated. The network communication in between is not further considered. Figure 97 visualizes the sequence diagram of the proposed co-simulation.

Initially, the consumer assigns instruction(s) to the co-simulation calling the **AssignInstruction** method. Since the **MInstruction** class can contain a further list of multiple instructions, it is also possible to provide several instructions as input. Internally, the co-simulation stores the instructions in a queue and returns a response whether the assignment was successful (**MBoolResponse**). Similar, to the MMUs, the co-simulation is also triggered by the **DoStep** function, which is cyclically called by the consumer. Within the **DoStep** routine, the co-simulation iterates over all MMUs according to its priorities and determines the next instruction to be executed. If the prerequisites of the instruction are fulfilled, the instruction is assigned to respective MMUs. Once the instruction is assigned, the MMU is called until it raises a finished event, or is externally aborted. The co-simulation furthermore, incorporates the results of the different MMUs. The result of the **DoStep** method is the incorporated result, represented as **MSimulationResult**.

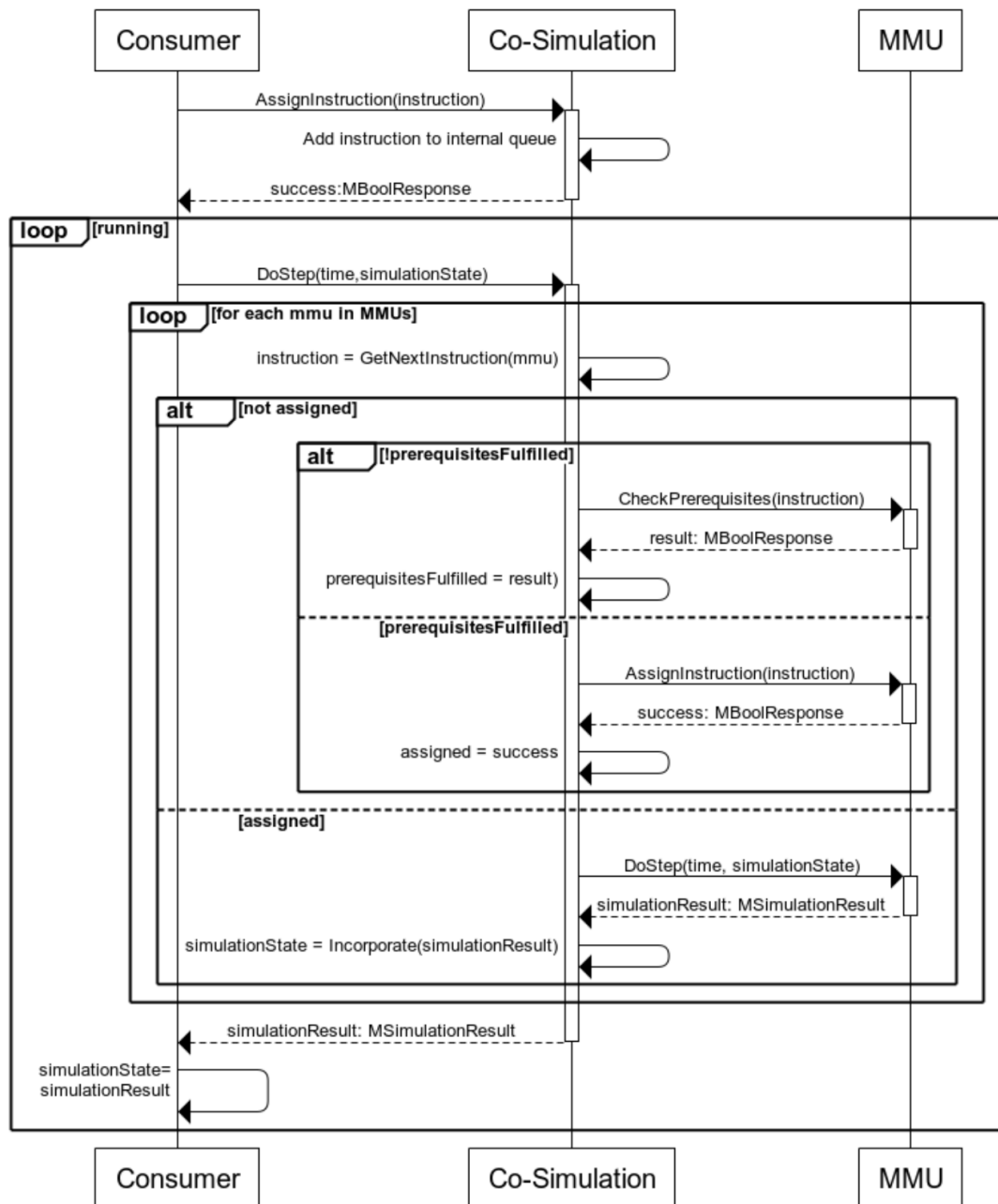


Figure 97 Sequence diagram for the co-simulation.

3.3.5 Behavior Execution

The Behavior Model & Execution Unit is an important mediator between the task description to be simulated and the final result, as it queries the state of the scene and decides based on this, which motions or MMUs are to be executed. Figure 98 visualizes the sequence diagram of the behavior execution workflow.

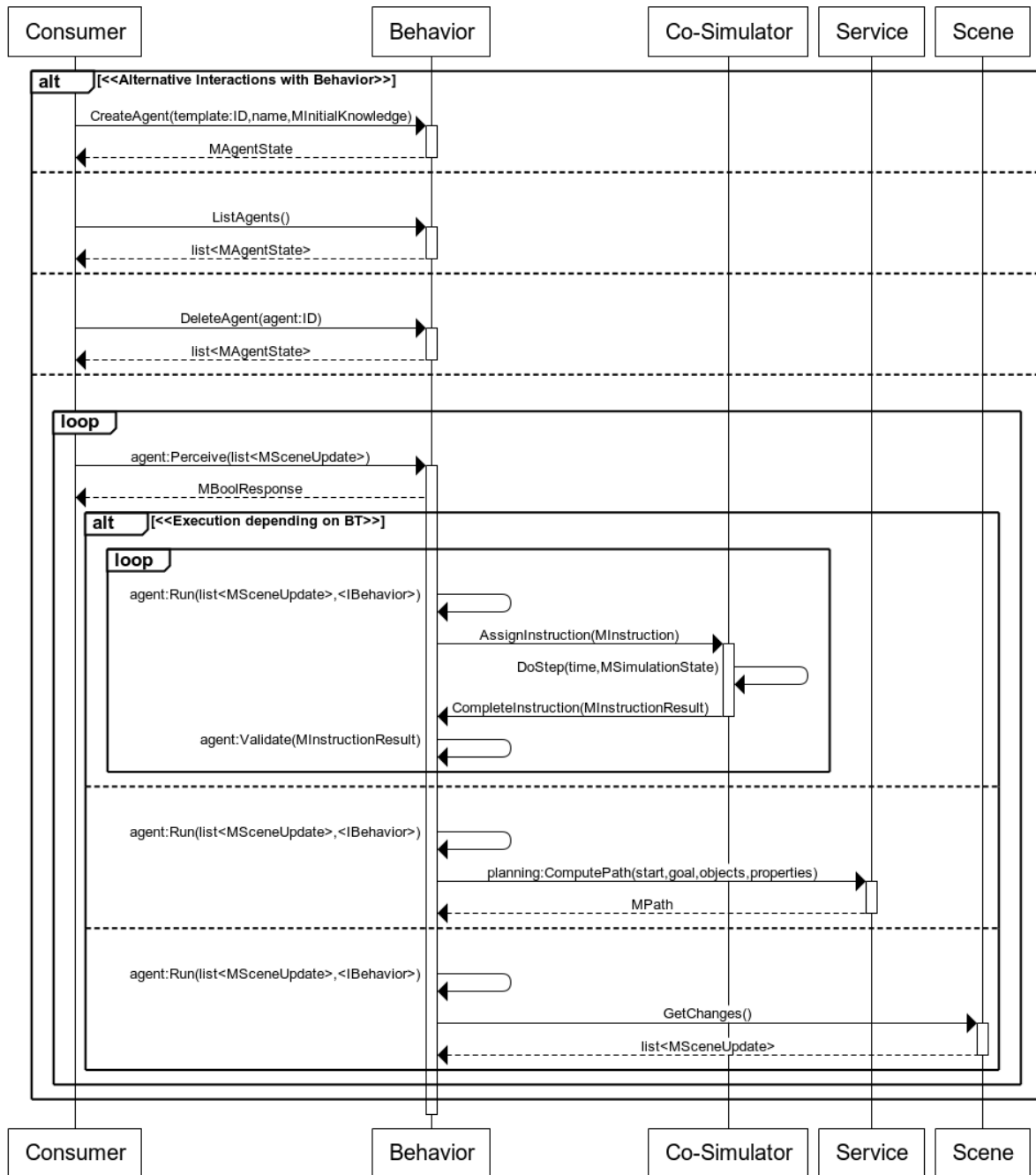




Figure 98 Sequence diagram of the behavior execution workflow.

As described in the previous chapter, a simulated worker is represented in our approach by a software agent. For the administration of software agents, i.e. the creation, deletion or execution of such agents, four endpoints are offered:

- **MBehaviorExecution:CreateAgent(MTemplate:ID,name,MInitialKnowledge)**: In order to create an agent, a reference to a predefined agent template is required, which contains, among other things, behavior models that the agent should use; a name; and the initial agent knowledge.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

This knowledge contains scene information and configuration data with which the agent can start with.

- *MBehaviorExecution:ListAgents()*: This function lists all agents.
- *MBehaviorExecution>DeleteAgent(MAgent:ID)*: This function deletes the agent with the submitted ID.
- *MAgent:Perceive(list<MSceneUpdate>)*: by sending scene updates to a particular agent, its reasoning cycle is activated because something in the scene has changed, and the agent may need to react. Internal BTs process this incoming information and decide how to deal with the new situation. These updates can be sent either event or game-tick based.

The reaction to new situations depends on the BT nodes used in a behavioral model. Either the updated agent knowledge is queried to check a particular situation and decide which behavior to execute later. Alternatively, new knowledge is generated based on the current situation or the current knowledge is updated. Besides these internal actions, the agent can also interact with other MOSIM components like the scene itself, the co-simulator or other services:

- *MBTScene:Run()*: If the agent decides in a specific situation to read out certain information from the scene that was not previously transmitted via the *MAgent:Perceive(list<MSceneUpdate>)* method, it is queried via the *MScene:GetChanges()* function.
- *MBTService:Run()*: Services should offer functions in MOSIM which can be used by different MMUs. They can also be used to support the decision-making process during Behavior Execution. For example, before generating instructions, the planned path planning service (*ComputePath(start,goal,objects,properties)*), could be used to determine whether a MoveTo-MMU can be executed with predefined parameters.
- *MBTAction:Run()*: For the generation of instructions (**MInstruction**) a special branch node (**MBTEvaluation**) is executed to evaluate the underlying sub-behaviors and finds out which n-actions are to be executed next. Action nodes (**MBTAction**) represent motions or MMUs with predefined constraints that are considered during the evaluation with a simulated future agent state. After submitting the instructions to the co-simulator by the executed **MBTAction**, the Behavior Execution waits for responses from the co-simulator in which the status (**MInstructionResult**) of the motion generation is notified. If a new situation arises that requires a different instruction or if a MMU in the previously determined instruction could not be executed, the Behavior Execution must re-plan a new **MInstruction**.
- *MBTAction:Validate(MInstructionResult)*: The first action in a generated **MInstruction** validates the incoming **MInstructionResult** to its state and then changes its own state so that a new **MInstruction** can be generated.

3.3.6 Target Engine

The target engine is an essential component of the proposed framework. In particular, the target engine contains the ground through scene and triggers the overall simulation. In the figure below a proposed workflow for the target engine is provided.

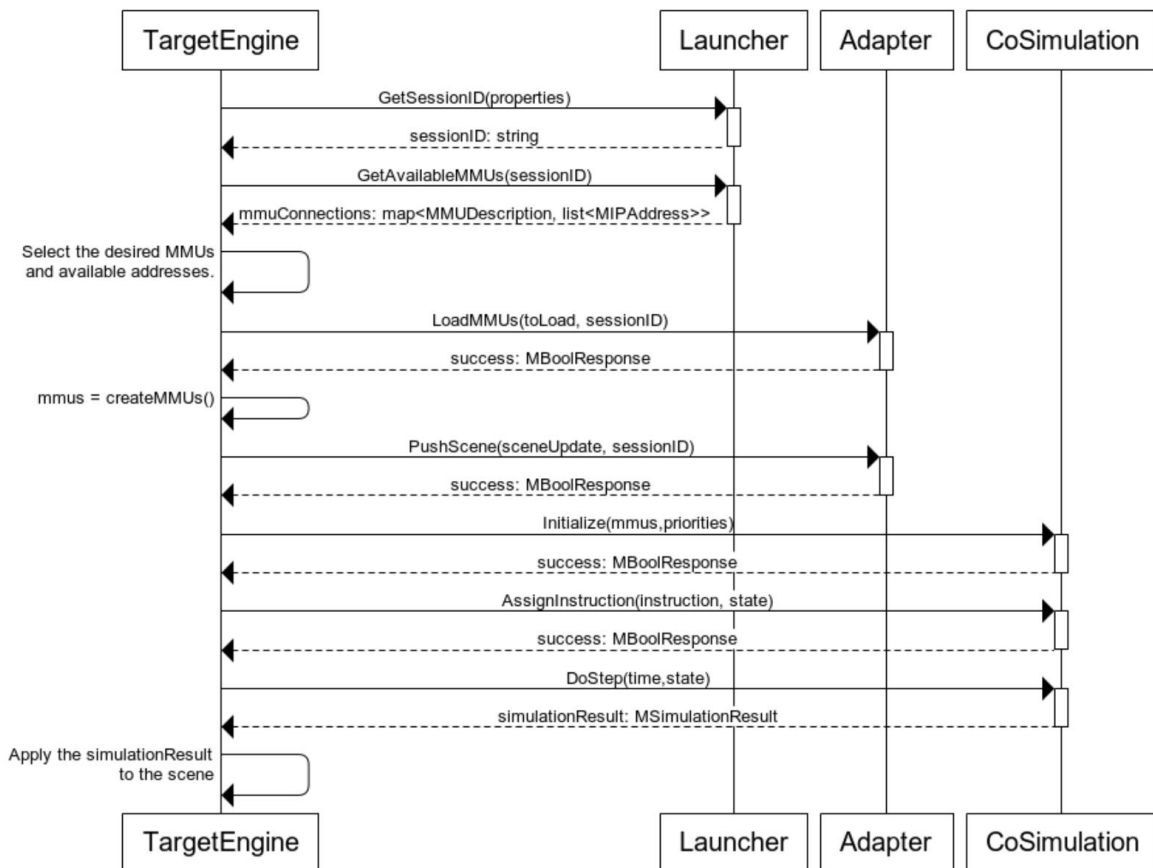


Figure 99 Sequence diagram illustration the workflow from a target engine perspective.

Initially, the target engine requests a unique session ID at the launcher/**MMIRegisterService**. Afterwards, the target engine can request the available MMUs with the associated connection properties from the **MMIRegisterService**. Next, the target engine internally selects the desired MMUs to be loaded. The target engine calls the **LoadMMUs** method of the respective adapter in order to load the desired MMUs. It is noteworthy, that a MMU might be available in multiple adapters. Therefore, it is up to the implementation of the target engine of which adapter to use.

After the MMUs have been successfully loaded, the target engine needs to provide the selected MMUs to the co-simulation. If the co-simulation is accessed using the proposed interface, a list of the unique id of the MMUs as well as its priorities need to be provided to the co-simulation. After the initialization of the co-simulation, the target engine can access its functionality. In particular, for each frame, the scene is pushed to each utilized adapter. The results of the co-simulation are applied to the scene at the end of each frame.

3.3.7 Skeleton Utilization

Despite the previously illustrated workflows, the interaction with the **MSkeletonAccess** depicts also a major aspect of the overall framework. Therefore, in the following an overview of the workflow of the skeleton utilization is provided.

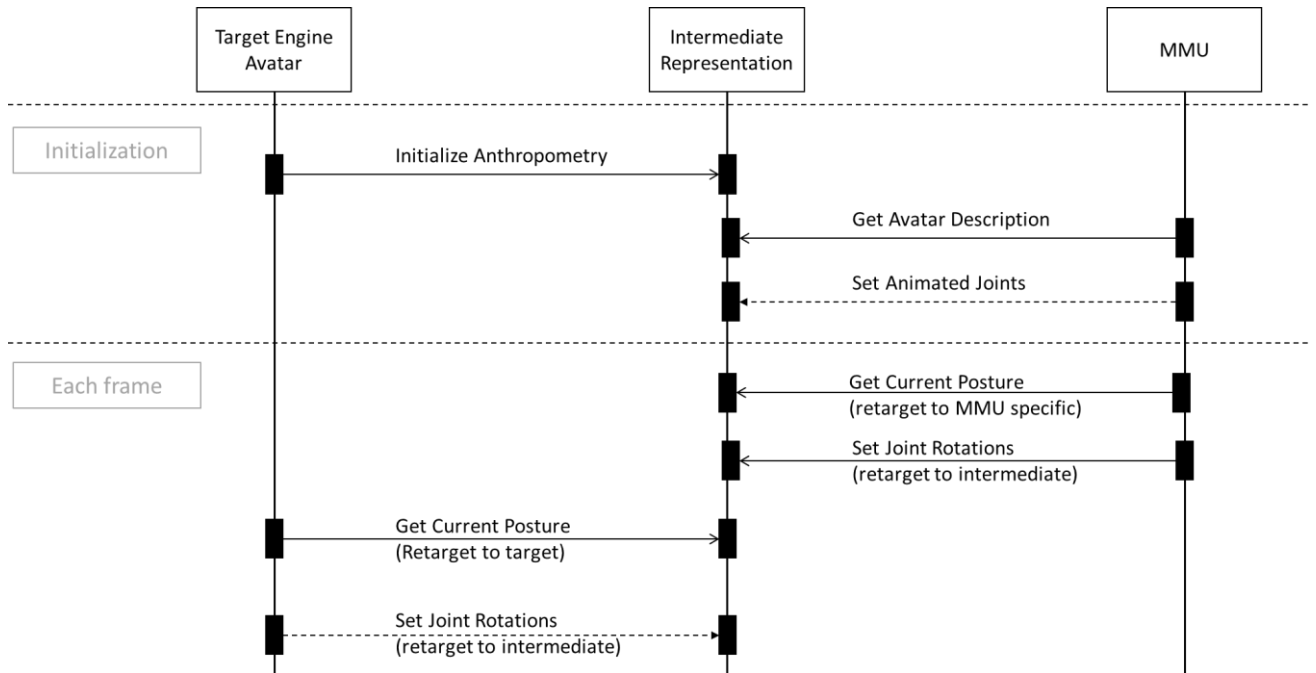




Figure 100 Sequence diagram illustration the workflow of the skeleton utilization.



	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

4 Summary and conclusions

Within the document, a first draft of the planned standard is proposed. In particular, the overall concept of the MOSIM framework was introduced. A detailed overview of the components and its interfaces was given. Moreover, the workflows of the components were described in detail. The document should serve as a base for utilizing and implementing the MOSIM framework.



Note that the document is considered as a living and dynamic document which is updated throughout the progress of the MOSIM project. In particular, the interfaces, formats and workflows, as described within this document are a first draft and are utilized at a starting point for the implementation phase of the project. However, feedback and gained knowledge during the implementation will be fed back to this document and the proposed framework will be adjusted. Especially with regard to the constraints, major adjustments are expected, since the corresponding task, which will focus on the design of constraints, has not started yet.

With the provided thrift files and the presented class diagrams in addition to the document, a first version of the MMI framework can be implemented.

	MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028 Project Coordinator: Thomas Bär, Daimler AG	
---	--	---

5 References

- [1] ITEA, "ITEA Project 07006 MODELISAR," [Online]. Available: <https://itea3.org/project/modelisar.html>. [Accessed 14 8 2019].
- [2] T. Blochwitz, T. Neidhold, M. Otter, M. Arnold, C. Bausch, M. Monteiro, C. Clauß, S. Wolf, H. Elmqvist, H. Olsson, A. Junghanns, J. Mauss, D. Neumerkel and J.-V. Peetz, "The functional mockup interface for tool independent exchange of simulation models," *8th International Modelica Conference 2011. Proceedings : March 20th-22nd, Technical Univeristy, Dresden, Germany*, 2011.
- [3] B. Wang and J. S. Baras, "Hybrid Sim: A modeling and co-simulation toolchain for cyber-physical systems," *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pp. 33-40, 2013.
- [4] B. Van Acker, J. Denil, H. Vangheluwe and P. De Meulenaere, "Generation of an optimised master algorithm for fmi co-simulation," *In Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pp. 205-212, 2015.
- [5] J. Bastian, C. Clauß, S. Wolf and P. Schneider, "Master for co-simulation using fmi," *Proceedings of the 8th International Modelica Conference*, pp. 115-120, 2011.
- [6] F. Gaisbauer, P. Agethen, M. Otto, T. Bär, J. Sues and E. Rukzio, "Presenting a Modular Framework for a Holistic Simulation of Manual Assembly Tasks," *Proc. of 51th CIRP Conference on Manufacturing Systems (CMS)*, 2018.
- [7] F. Gaisbauer, P. Agethen, T. Bär and E. Rukzio, "Introducing a Modular Concept for Exchanging Character Animation Approaches," *Proc. of Eurographics*, 2018.
- [8] F. Gaisbauer, J. Lehwald, P. Agethen, J. Sues and E. Rukzio, "Proposing a Co-simulation Model for Coupling Heterogeneous Character Animation Systems," *14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP)*, 2019.
- [9] I. Zinnikus, A. Antakli, P. Kapahnke, M. Klusch, C. Krauss, A. Nonnengart and P. Slusallek, "Integrated Semantic Fault Analysis and Worker Support for Cyber-Physical Production Systems," *2017 IEEE 19th Conference on Business Informatics (CBI)*, vol. 01, pp. 207-216, 2017.
- [10] A. Antakli, E. Z. I. H. D. Herrmann and K. Fischer, "Intelligent Distributed Human Motion Simulation in Human-Robot Collaboration Environments," *IVA '18: International Conference on Intelligent Virtual Agents. International Conference on Intelligent Virtual Agents (IVA-2018), November 5-8, Sydney, New South Wales, Australia*, 2018.
- [11] Apache, "Apache Thrift," [Online]. Available: <https://thrift.apache.org/>. [Accessed 13 8 2019].

	<p style="text-align: center;">MOSIM End-to-end Digital Integration based on Modular Simulation of Natural Human Motions ITEA 3, 17028</p> <p style="text-align: center;">Project Coordinator: Thomas Bär, Daimler AG</p>	
---	--	---

[12] BML, "Behavior Markup Language Website," [Online]. Available: <http://www.mindmakers.org/projects/bml-1-0/wiki>. [Accessed 13 8 2019].

[13] S. Kopp, K. B. S. Marsella, A. Marshall, C. Pelachaud, H. Pirker, K. Thorisson and H. Vilhjalmsón, "Towards a common framework for multimodal generation: The behavior markup language," *Proc. of Intelligent Virtual Agents (IVA'06)*, 2006.

6 Appendix

6.1 Exemplary MMU Description File

```
{
  "Name": "UnityLocomotionMMU",
  "ID": "4b8e33c0-2db1-424a-96a8-ca9fdcbcb56a",
  "AssemblyName": "UnityLocomotionMMU.dll",
  "MotionType": "walk",
  "Language": "UnityC#",
  "Author": "Felix Gaisbauer, Daimler AG",
  "Version": "1.0",
  "SupportedProportions": {},
  "Properties": {},
  "Dependencies": ["unitylocomotionmmu"],
  "Events": [],
  "LongDescription": "Implementation of a locomotion MMU which actually models walking from point a to point b. The MMU uses the path planning service.",
  "ShortDescription": "Data driven locomotion MMU based on the Unity Engine.",
  "Parameters": [
    {
      "Name": "TargetID",
      "Type": "string",
      "Description": "The id of the target object/transform. Either Target ID or Target Name must be defined.",
      "Required": true
    },
    {
      "Name": "TargetName",
      "Type": "string",
      "Description": "The name of the target object/transform. Either Target ID or Target Name must be defined.",
      "Required": true
    },
    {
      "Name": "ReplanningTime",
      "Type": "int",
      "Description": "The timespan [ms]after which the replanning is performed.",
      "Required": false
    }
  ]
}
```



```
"Name": "Velocity",
  "Type": "float",
  "Description": "The velocity of the avatar [m/s].",
  "Required": false
},
{
  "Name": "TransitionModeling",
  "Type": "bool",
  "Description": "Specifies whether the transition modeling is enabled.",
  "Required": false
},
{
  "Name": "Trajectory",
  "Type": "list<MVector2>",
  "Description": "An optional trajectory which can be defined.",
  "Required": false
}
],
"__isset": {
  "SupportedProportions": true,
  "Properties": true,
  "Dependencies": true,
  "Events": true,
  "LongDescription": true,
  "ShortDescription": true,
  "Parameters": true
}
}
```

6.2 Exemplary file of the Skeleton

```
HIERARCHY
ROOT PelvisCentre
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xoffset Yoffset Zoffset Xrotation Yrotation Zrotation
}
```



```
JOINT S1L5joint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xrotation Yrotation Zrotation

  JOINT T12L1joint
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Xrotation Yrotation Zrotation

    JOINT T1T2joint
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Xrotation Yrotation Zrotation

      JOINT C4C5joint
      {
        OFFSET X Y Z
        ROTATION W X Y Z
        CHANNELS Xrotation Yrotation Zrotation

        JOINT Headjoint
        {
          OFFSET X Y Z
          ROTATION W X Y Z
          CHANNELS Xrotation Yrotation Zrotation

          JOINT MidEye
          {
            OFFSET X Y Z
            ROTATION W X Y Z
            CHANNELS Xrotation Yrotation
          }
        }
      }
    }
  }
  JOINT LeftShoulder
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Xoffset Yoffset Zoffset Xrotation Yrotation Zrotation

    JOINT LeftElbow
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Yrotation Zrotation

      JOINT LeftWrist
      {
        OFFSET X Y Z
        ROTATION W X Y Z
        CHANNELS Xrotation Yrotation Zrotation

        # Finger Joints
      }
    }
  }
  JOINT RightShoulder
```



```
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xoffset Yoffset Zoffset Xrotation Yrotation Zrotation

  JOINT RightElbow
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Yrotation Zrotation

    JOINT RightWrist
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Xrotation Yrotation Zrotation

      # Finger Joints
    }
  }
}
JOINT RightHip
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xoffset Yoffset Zoffset Xrotation Yrotation Zrotation

  JOINT RightKnee
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Yrotation Zrotation

    JOINT RightAnkle
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Xrotation Yrotation Zrotation

      JOINT RightBall
      {
        OFFSET X Y Z
        ROTATION W X Y Z
        CHANNELS Xrotation Yrotation Zrotation
      }
    }
  }
}
```

```
...
JOINT LeftWrist
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xrotation Yrotation Zrotation

  JOINT LeftThumbMidcarpalJoint
```

```
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

  JOINT LeftThumbMetacarpophalangealJoint
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Xrotation Yrotation

    JOINT LeftThumbCarpalInterphalangealJoint
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Yrotation
    }
  }
}
JOINT LeftIndexMidcarpalJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

  JOINT LeftIndexMetacarpophalangealJoint
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Xrotation Yrotation

    JOINT LeftIndexCarpalProximalInterphalangealJoint
    {
      OFFSET X Y Z
      ROTATION W X Y Z
      CHANNELS Yrotation

      JOINT LeftIndexCarpalDistalInterphalangealJoint
      {
        OFFSET X Y Z
        ROTATION W X Y Z
        CHANNELS Yrotation
      }
    }
  }
}
JOINT LeftMiddleMidcarpalJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

  JOINT LeftMiddleMetacarpophalangealJoint
  {
    OFFSET X Y Z
    ROTATION W X Y Z
    CHANNELS Xrotation Yrotation

    JOINT LeftMiddleCarpalProximalInterphalangealJoint
    {
```



```
OFFSET X Y Z
ROTATION W X Y Z
CHANNELS Yrotation

JOINT LeftMiddleCarpalDistalInterphalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation
}
}
}
}
JOINT LeftRingMidcarpalJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

JOINT LeftRingMetacarpophalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xrotation Yrotation

JOINT LeftRingCarpalProximalInterphalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

JOINT LeftRingCarpalDistalInterphalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation
}
}
}
}
JOINT LeftLittleMidcarpalJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

JOINT LeftLittleMetacarpophalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Xrotation Yrotation

JOINT LeftLittleCarpalProximalInterphalangealJoint
{
  OFFSET X Y Z
  ROTATION W X Y Z
  CHANNELS Yrotation

JOINT LeftLittleCarpalDistalInterphalangealJoint
{
  OFFSET X Y Z
```



MOSIM
End-to-end Digital Integration based on Modular
Simulation of Natural Human Motions
ITEA 3, 17028



Project Coordinator: Thomas Bär, Daimler AG

ROTATION W X Y Z
CHANNELS Yrotation

}
}
}
}
}
...
}