ARCHIVE OF MECHANICAL ENGINEERING

DOI: 10.24425/ame.2022.141515 2022, Vol. 69, No. 3, pp. 345–373

Adam KŁODOWSKI[1], Ilya KURINOV[1], Grzegorz ORZECHOWSKI[1], Aki MIKKOLA[1]

Simulating human motion using Motion Model Units – example implementation and usage

Received 1 February 2022, **Revised** 19 April 2022, **Accepted** 28 April 2022, **Published online** 30 August 2022

Keywords: motion generation, example with source code, MMU, MOSIM, human simulation

Human motion is required in many simulation models. However, generating such a motion is quite complex and in industrial simulation cases represents an overhead that often cannot be accepted. There are several common file formats that are used nowadays for saving motion data that can be used in gaming engines or 3D editing software. Using such motion sets still requires considerable effort in creating logic for motion playing, blending, and associated object manipulation in the scene. Additionally, every action needs to be described with the motion designed for the target scene environment. This is where the Motion Model Units (MMU) concept was created. Motion Model Units represent a new way of transferring human motion data together with logic and scene manipulation capabilities between motion vendors and simulation platforms. The MMU is a compact software bundle packed in a standardized way, provides machine-readable capabilities and interface description that makes it interchangeable, and is adaptable to the scene. Moreover, it is designed to represent common actions in a task-oriented way, which allows simplifying the scenario creation to a definition of tasks and their timing. The underlying Motion Model Interface (MMI) has become an open standard and is currently usable in MOSIM framework, which provides the implementation of the standard for the Unity gaming engine and works on implementation for the Unreal Engine are under way. This paper presents two implementation examples for the MMU using direct C# programming, and using C# for Unity and MOSIM MMU generator as a helping tool. The key points required to build a working MMU are presented accompanied by an open-source code that is available for download and experimenting.

✉ Adam Kłodowski, email: adam.klodowski@lut.fi

[1]Department of Mechanical Engineering, LUT University, Yliopistonkatu 34, 53850 Lappeenranta, Finland. Emails: adam.klodowski@lut.fi, ilya.kurinov@lut.fi, grzegorz.orzechowski@lut.fi, aki.mikkola@lut.fi

Abbreviations

DHM – Digital Human Modeling
MMI – Motion Model Interface
MMU – Motion Model Unit

1. Introduction

According to Eurostat, the manufacturing sector employed more than 28.5 million people in the EU in 2017 [1]. Product development time in the last decade has been significantly reduced from 48 to 25 months [2]. Shortening product development cycle puts pressure on production workers, who need to adapt fast for the new tasks. The number of people and robots included in the modern production process requires close cooperation between the two entities in a safe way. Untested robot codes or badly planned human workstations can result in injuries, production inefficiencies and quality problems. To give production planning specialist more time during such an already shortened product introduction cycle, it is important to start simulating future production lines in the early stages [3]. Currently available tools allow one to focus on either the automated sections of the production chain or on a single worker. Moreover, such simulation tools are often incompatible with each other and therefore, advanced analyses are time-consuming and can only cover small critical sections of the production process.

Accordingly, it is important to bridge the gap between production planning and product design covering both industrial machines and human workers in a single simulation. Inclusion of a human to the assembly process simulation offers several benefits. First of all, timing of the action can be assessed reliably for new tasks. Secondly, interaction between human workers and machines can be evaluated. Thirdly, bottlenecks and accessibility problems can be identified, and finally, workplace ergonomics can be reviewed and optimized at the production cell design stage. A virtual training environment for production workers can also be created before the actual production line is commissioned. This can significantly shorten the ramp-up time when new products are introduced. Discussions with worker's unions and authorities can be based on visual feedback on the production cell ideas. Finally, instructional assembly videos can be created and distributed to workers and their supervisors ahead of production start [4].

The new simulation framework is based on standardized interfaces to ease changing simulation engine, visualization tools, as well as integration of data exchange plugins to specialized Product Life cycle Management (PLM) or Computer Aided Design (CAD) systems utilized by industry. Data exchange and core framework components are built in an open-source spirit to ease the integration of tools and services with the framework.

1.1. Human modelling platforms

Digital human modeling is a well-known practice in industry and science [5–7]. Digital human modeling was used to investigate the effects of the environment on the human. For example, the first human model was developed in 1960 to simulate the effect of zero gravity on the human body in space. In 1967, an anthropomorphic human model was created to check the ergonomics of aircraft cockpits [8].

In 1985, the model TEMPUS was created, which allowed for analyzing the movement process of the digital human. The TEMPUS model has been the predecessor to the modern JACK model that is widely used nowadays. The JACK model was designed for product ergonomics analysis, but it can also be used for other purposes through the interface. The simulation procedure is equipped with modules allowing the application of the simulated human in factory designing and processes planning [9]. The JACK inherits robotics methods for simulation of movements and does not use any motion blending for motion transitions, consequently producing unnaturally-looking motion sequences [8].

During the 1980's the SAFEWORK model was developed at the École Polytechnique [10]. The SAFEWORK consists of three modules covering anthropometry, movement, and analysis. The model is focused on workplace design and work processes planning. The movements of the digital human model are obtained via inverse kinematics. The analysis tool allows for vision simulation, fixed accessibility areas, joint-comfort analysis, and calculation of maximum forces within the joints. This technology is currently available as an integration unit in the CATIA CAD software [8].

Another digital human modeling tool is the Santos [11], which originates from the Virtual Soldier Program started in 2003 at the University of Iowa. The technology focuses on human performance in task-specific training. This technology considers human characteristics such as strength, fatigue, flexibility, balance, vision, and posture, as well as clothing and other equipment worn on the body, external forces, and environmental conditions. The Santos human model has 100 degrees of freedom enabling more realistic behavior.

In 2002, the Biomechanics Research Group, Inc. launched the LifeMOD Human Modeler plugin for the MSC Adams [12] general purpose multibody dynamics simulation software. The LifeMOD plugin allowed the integration of the human skeletal system into the simulation and focused on the muscular and joint dynamics. It covered PID-controlled muscle models, including tendons as well as the Hill muscle model. The software had been developed for several years until 2012, when the software company was taken over by Smith & Nephew, Inc. At that time, the software became unavailable to researchers and the public as the new owner focused on providing services rather than software packages [8].

In 2007, the OpenSim 1.0 was released. Today, the OpenSim is one of the most popular platforms for simulating human motion, muscle activation, or optimization of human balance stabilization during physical activity. It is one of the platforms that became widely adopted by researchers due to the ease of numerical models integration and open-source nature of the project allowing for the customization of the code as needed [13]. Although the capabilities of the software are remarkable and the user community is large, it is not well suited for commercial use as its user interface is far from being user friendly, modeling with OpenSim is time-consuming, and the support is only available through a forum. Nevertheless, it is one of the projects that enable the creation of research bases for commercial human simulation platforms. As of January 2022, the Google Scholar indexed over 12 000 articles related to the OpenSim.

Another open-source platform worth mentioning is the HuMAnS Toolbox by INRIA. It focuses on the simulation of humanoid robots, but is also usable for human simulation. The analysis of human motion stability and related ground contact is one of the core focuses of the HuMAnS package [14]. This project was a response to the need for a simulation platform for humanoid robots that would be capable of evaluating the stability of such robots during locomotion, offering ease of integration of additional models as required by the user. The lack of tools satisfying this need, and its closed source nature makes it difficult to integrate commercial solutions to drive this project forward.

Among commercially available solutions for human simulation one can find the Ramsis by Human Solutions [15], the Tecnomatix Jack by Siemens [16], the AnyBody by AnyBody Technology A/S [17]. Ramsis and Tecnomatix Jack packages focus mostly on ergonomics, while the AnyBody is more muscle and joint simulation oriented. Furthermore, the Simcenter Madymo [18] is one of the popular solvers for automotive occupants and pedestrian safety evaluation. The Madymo features multibody, finite element, and computational fluid dynamics solvers that allow for comprehensive studies of crash test dummies in virtual accident simulations as well as active human models that can be used in pre-crash situations where human reactions must be considered.

It can be concluded from the state of the art analysis that human simulation is mostly considered as being in a domain separate from factory simulation and worker simulation, in general. Ergonomics is one of the driving forces of implementing human simulations in modern industry, but is also affected by safety concerns and the need for manual work feasibility analysis before commissioning of the plant. These targets are difficult to merge in a single simulation environment due to a lack of widely adopted standards for the human model exchange and limitations in general purpose simulation codes. On the other hand, highly specialized software packages can solve ergonomic problems or evaluate the safety of human operations, but tend to overlook details of machine implementations. This situation exists mostly due to closed source software distribution models dominating the commercial market. A single company may not be capable of offering solutions that would be both very general and simple to use at the same time, allowing for quick simulation model creation and validation. Experts in the human musculoskeletal simulations utilize platforms such as OpenSim, AnyBody, and LifeMOD. Ergonomic experts prefer Tempus, Jack, Safework, or Ramsis, while plant designers utilize different software families. Combining results of multiple software packages to obtain a coherent vision of the problem is difficult, and combining different file formats utilized in closed-source software packages can be a real showstopper [19].

1.2. Motion Model Units

The Motion Model Units (MMU) concept as an extension to the Functional Mock-Up (FMU) [20] standard aims at solving the compatibility problem and allow experts from motion and human simulation to provide simple building blocks for various software packages using one plug and play software implementation [19]. Currently, MMU development is focused on providing basic human motion generation blocks that can be used in most common industrial applications covering production, maintenance, but also crowd or pedestrian behavior for the use in autonomous vehicle studies.

The concept of dividing motion into blocks that can be repeated and merged into different configurations is one of the common techniques utilized in games, where motion blending combined with a state machine is used to smoothly transition from one motion to another. For example, the default state is idling, where the avatar performs pseudo-random body movement, occasionally performing some operation like weight balance shift from one leg to another or head motions. When an input is received or another trigger action is initiated, the state machine transitions from the idle motion to, for example, walking and loops over the walking pattern until a stop command is issued or transition to, for example running, happens. Such a state machine combined with motion blending is one of the popular components in many game development tools like the Unity [21] or the Unreal Engine [22]. It is a game developer's job to provide state machine diagrams, define transition triggers, and prepare animations that can blend for realizing a constrained set of actions. It is important to note that this concept handles only motion and does not directly include interactions with the environment. Such interactions need to be programmed individually in every case.

MMUs represent individual motions combined with logic blocks that deal with the current scene state and determine how the MMU is going to perform its function, and if it is able to fulfil its function at all [23]. This means containerization of motion and logic in a single plug-and-play package. In the MOSIM framework [24], the logic can be placed on two levels. On the first level, the behavior modeling, target actions are derived based on high-level tasks (goals) and scene states are considered [23]. On this level, selection of the MMU happens, and a strategy of low-level actions instantiating is implemented. The second level is the MMU level, where the fine-grain logic is coded. The logic contained in the MMU deals with adjusting motion output based on target avatar anthropometry, evaluating motion constraints provided by the behavior and environment, and adapting the motion within the motion type that the MMU implements.

This adaptation, where a human skeleton is modeled to resolve joint constraint dependencies, is a typical biomechanical problem. In most cases, the multibody model is limited to the inverse kinematics of the specific body segment to achieve real-time efficiencies. Examples would include a hand reaching action or both legs moving for locomotion.

To develop an MMU targeted at optimizing movement ergonomics, a detailed musculoskeletal model could be employed to provide motion information about joint loading and muscle forces at the expense of simulation speed. When simulating the assembly of tight fit connections, for example, a muscle-driven model could be used to determine how feasible it is to make the connections using bare hands or if specific tools are required.

Carrying large objects, interacting with push carts, or stacking objects could result in a loss of balance, and therefore for such scenarios, a multibody model should be integrated with the MMU to properly reflect interactions between the human and the manipulated objects instead of relying entirely on kinematic manipulation of the scene objects. In a simulation, the objects subject to interaction are considered rigid bodies with at least basic inertial properties and contact surfaces defined. An avatar walking into an obstacle or moving one object such that it collides with another must impact the simulation result. This behavior is controllable through the user interface, and if desired, certain objects can be simplified to just geometrical representations without physical representation in the multibody simulation engine.

While the MMU is a separate part of the code, it is a target visualization engine independent and can be reused in different target engines. This allows for preparing not only motion but also the logic behind it to be bundled and made available for a large group of developers in a single package.

Blending of motion output by subsequent MMUs can be performed by a co-simulator – which can use behavior to plan ahead of the output to perform blending between start and end frames provided by the MMUs. However, this is challenging as the initial MMU pose might not be well defined. The second option is blending performed by the MMU, which uses the output of the previous MMU as the input and if this avatar state is not optimal, it can blend it towards the desired start state [23].

Finally, a separate specialized MMU can be created to serve purely as blending entity, combining the initial state of the next MMU with the final state of the previous MMU. Initial and final states of the MMU avatar poses could also be standardized within a group of MMUs, alleviating the need for motion blending between consecutive MMUs. This strategy works if MMUs are only run sequentially, but fails when multiple MMUs are run in parallel modifying the avatar's pose. This paper focuses on addressing issues of human motion utilized in factory planning, which might affect the overall results of the simulation process. The objective of this paper is to introduce the application of an example code procedure to control a digital human model using motion model unit implementation. The introduced control procedure is based on motion capture and linear motion blending techniques commonly used in the gaming industry.

2. Human model implementation

2.1. Challenges

Human avatars as three-dimensional (3D) models can be introduced to practically any simulation environment that provides 3D visualization. Bringing avatars to life is, however, the bottleneck of many simulation workflows. General simulation tools usually allow imports of 3D geometry in one of the standard data exchange formats, but representation of human joints and thus mechanical connections is already the first challenge. The second challenge is the actuation of the human joints and relative movement of the avatar in the simulation environment. While simple stationary motion can be prescribed using digitally generated data, to achieve naturally-looking articulation, motion capture data is often employed. This implies that for every distinct action, a separate motion capture is needed. Such an approach works well for repetitive motions, like walking, posture change, making gestures, replicating face expressions and other similar motions.

Actions that diverge considerably in kinematics depending on the environment are not well suited for a motion capture-based approach. These are, for example, reaching a point or moving an object. Those actions are dependent on the relative position of the human, and the manipulated object and the number of these combinations of parameters is infinite. In such tasks, a better-suited approach relies on the direct control of the avatar motion, for instance using a virtual reality environment, where a user wearing a motion suit is able to directly move the avatar in the simulation and interact with the objects being simulated.

Another option is the usage of inverse kinematics for motion planning. Such an approach allows for the elimination of the need for preparation of motion data for every single use case, as the motion data is generated on a need basis. Reaching, grasping, and object manipulation are good examples where this method can be a robust solution. The main drawback of the inverse kinematic approach is the ergonomics and the often unnatural motions. As inverse kinematics looks for any possible solution not necessarily optimal from an ergonomics perspective, in many cases it has to be supported by an optimization routine that is capable of weighing possible solutions to select one that is both anatomically feasible and natural-looking, where the latter is more cumbersome to define mathematically [25].

That implies that significant effort is required for the implementation of such systems in general-purpose simulation software. In many cases, it can be an inefficient solution, as physical testing could be accomplished in the same amount of time. Therefore, for practical reasons, motion generation for the human avatar is often a blend of the above-mentioned approaches. Balancing between algorithms allows for a compromise between robustness and natural-looking motion. As a generation of human motion can be considered to be a separate field of science requiring broad expertise, it is desirable to modularize the human motion generation process, so that specialized companies could provide ready building blocks for the simulation end-users. The building blocks can be interchanged and sequenced in any desired motion scenario. This is where the Motion Model Units (MMU) concept has been introduced. Encapsulation of the MMU in a standard package enables research institutes and companies that produce MMUs and the simulation framework end-users who utilize the MMUs to connect with each other in their virtual testing environments.

Once motion has been generated for simple tasks using MMUs, the complex motion can be created by running several MMUs sequentially and in parallel. For example, pick and place actions can be created from a series of walking, reaching, grasping, and positioning focused MMUs. The sequence could be: walking to the object, reaching for the object while running simultaneously, idle motion of the body, then additionally parallel execution of grasping followed by carrying (attachment of the object to the hand), continued with another walk action keeping only carry motion active during the walk, finally, performing positioning action, followed by release and idle motion. Even this simple example requires a mix of parallel and sequential operations that need to be synchronized with each other. In parallel blocks, the hierarchy of execution is important in terms of the MMU motion overriding by hierarchically higher placed MMUs. This presents another challenge in human simulation – namely, motion synthesis and motion planning that also appears in simple animation-driven models without the MMU implementation concept.

2.2. Behavior modeling

Having an avatar in the simulation capable of performing basic actions is still far from an optimal solution, as it requires information on when and which MMU to use and what parameters should be provided to the MMU so that it would execute desired motion taking into account the start and stop constraints. This part is covered by the behavior modeling of the MOSIM framework. In behavior modeling, complex tasks like pick and place actions are broken down into basic actions that can be directly mapped onto the MMUs. Definition of high-level tasks and the associated task breakdowns is something that needs to be done only once and can be reused in multiple simulations. The reasoning process that links the high- and low-level tasks together requires information about the available MMUs, and the simulation environment state before and after each MMU is executed. From the end user’s perspective, behavior modeling allows for a simplification of human modeling to simple task sequence definition that is automatically converted to fully parametrized MMU sequences, that in turn produce desired motions and scene manipulation set of actions. It is a big step forward in user experience as behavior modeling, and the MMU concept eliminates the need for programming and motion capture overhead.

3. Motion model interface

The motion model interface is a modular system consisting of motion model units. A Motion Model Unit is a block containing the motion provider of the digital human model. The way in which the motion is generated within the MMU is irrelevant for the MMI standard. The standard covers interface and a set of requirements towards software code to be usable as the MMU. The motion generator of the MMU creates motions needed for the desired application and should be constrained to well defined actions. The motion can be represented as a skeleton configuration provided for every time step. Alternatively, MMI standard allows MMUs to provide only a limited number of joint angles that the MMU modifies during the runtime. This allows simplification of the MMU code for actions that do not involve the whole body motions. Motion generation can be as simple as replaying animation sequences defined as body poses change over time. It can also be based on inverse kinematics, forward dynamics simulation of the human or artificial intelligence. The limit is just imagination of the MMU developer.

MMUs are run hierarchically, which means that an MMU of a higher level is run after a lower level one, receiving the output pose of the lower level MMU as the input, and providing a modified pose as the output. It is expected that as the standard reaches a wider adoption, the most common implementation would utilize motion blending of motion capture data. This is mostly due to simplicity and good real-time performance; statistical approaches reinforced with the AI for covering complex scenarios where animation smoothness should be maintained and versatility of the model is more important than the computational cost. Finally, specialized MMUs will be aiming at ergonomics optimization as this is in high demand from industry, based on the number of companies offering human simulation packages focused on ergonomics.

In the MOSIM framework, the commonly used code can be excluded from the MMU and repackaged as a service, allowing reuse in multiple MMUs and other framework components.

3.1. MMU background

MMUs utilize adapters to run them. This way almost any programming language can be used to create MMUs, provided that an adapter is available for such a language. Adapters can be seen as the middleman between the framework and the MMU; they are responsible for loading MMUs and providing buffered scene data access and a communication layer between the MMU and the framework (mostly the target engine), which could be running on a remote machine. In this context, MMUs are dynamic link libraries. In special cases, an MMU can be a standalone software in an executable format, then adapter functionality can be built into the MMU. This is a less resource efficient solution as multiple MMUs will hold a complete copy of the scene, which has to be synchronized frequently while using the adapter only. The adapter holds a copy of the scene, and the adapter only needs to synchronize it with the scene.

MMUs communicate with the framework using the Apache Thrift [26], which simplifies the integration of multiple programming languages into a cooperating system and provides a skeletal code in selected programming languages based on a simple protocol definition. The Thrift protocol implements remote procedure calls, which simplifies the programming of the communication protocol included in the Thrift library as a generic application agnostic code. The MMU runs the server socket module listening for incoming connections. Once the connection is established (most commonly from target engine components), an MMU can receive commands and provide a response for every time step on request. The response for every time step is requested from the MMU by the client, providing scene information and simulation time as parameters. An MMU can also establish client connections to other MMUs to use them as submodules or call services to utilize their routines. On startup, the MMU connects to the MMI register host to register its state and communicate its availability and capabilities to the framework components. All framework components can receive information about available MMUs and services from the register server. MMUs and services, therefore, must register themselves at the MMI register to be discoverable by the clients. Communication protocol supports basic functions that the MMU requires, including sending instructions for the MMU to restart or shutdown itself.

On initialization call, the MMU receives avatar information, including its dimensions and pose. The MMU also receives constraints applied to the action objects or work space where the MMU is responsible for interpretation and obeying. The MMU can raise one of the standard events in response to situation change in the scene or to signalize its completion, or it reached an unsolvable situation that resulted in an error. The events are used for MMU synchronization.

3.2. Event types

Event types are described by the string parameter, they are, however, constraint to predefined values that represent semantically understandable events that the MMU can raise or receive. Currently supported events can be checked at api.mosim.eu. This is a permanent place where up-to-date living MOSIM standards will be kept and made available. The result is provided in the JSON format, that allows automatic parsing by software tools. Events are used to signal specific states of the MMU or events in the scene that can be considered by MMUs. Currently, there are eight supported events: `start`, `ready`, `stroke_start`, `stroke`, `stroke_end`, `end`, `abort`, `warning` and `exception`.

3.3. Constraint types

Eight constraint types are supported by the framework: Geometry, Velocity, Acceleration, Path, Joint Path, Posture, Joint. Additionally, a dictionary of properties can be used to describe semantic parameters that should be associated with scene objects. It is important to note that such parameters have to be understandable to a behavior reasoning engine if it is supposed to use them, and by MMUs that should utilize such parameters. Therefore, the web page at api.mosim.eu offers a data list of available parameters and their meaning.

Geometry constraint has only one required parameter – the parent Object ID, which specifies the base coordinate system reference point for the constraint. The remaining parameters are optional: the transform parameter covers rotation and translation of the local coordinate system with respect to the parent object coordinate system; the translational constraint represents limits along all three axes that can be used to define either extremes of a box or ellipsoid space. Similarly, rotational parameters define limits for x, y, z rotations in degrees. Finally, the weighting factor that takes values from 0 to 1 defines the importance of the constraint. The weighting factor can be used to describe sets of constraints that might be mutually exclusive. Then the weighting factors should be used to find an optimal solution that best fulfills the constraints with the higher weighting factor values. The weighting factor of 1 should always be interpreted as the constraint that must be fully satisfied, and if it is impossible, the simulation should fail. The translation and rotation parameters are expressed in [m/s] and [rad/s], respectively.

Velocity constraints allow one to define, for example, the initial velocity (both linear and rotational), the maximum or minimum velocity value, or simply the target velocity value. In the same manner as the geometrical constraint, velocity constraints require specification of the parent object ID and allow for specifying transformation of the parent to local coordinate transform.

The acceleration constraint is defined analogically to the velocity constraint, and its values are expressed in [m/s^2] and [rad/s^2] for translation and rotation values, respectively. The path constraint is a list of geometrical constraints forming a series of values that can be interpreted, for example, as the walk path, the motion path, or transition states. The weighting factor can also be defined there.

The joint path constraint is a list of sequential values of geometrical constraints applied for the selected avatar's joint. This type of constraint can be mostly used to store motion data for a single avatar's joint. The posture constraint is a set of joint angles uniquely defining a single avatar's posture. In addition, an optional parameter list of joint constraints can be supplied. The joint constraint is defined by 4 parameters: joint type, geometry constraint, velocity constraint, and acceleration constraint. The joint type is simply an indication of a specific human joint, while the remaining constraints follow the definitions shown at the beginning of this section.

3.4. Constraint interpretation

Constraints IDs are used not only to refer to the constraints but also to give them semantic meaning. This is why standard definitions and a list of MMUs and their parameters are available from mosim.eu. Then, each developer who wants to build a new MMU performing action that has already been defined, could reuse those definitions to create fully compatible MMUs with the existing scene objects. The aim of preparing a set of standardized definitions is that they will be suggested as constraint default semantics that the developers should use. This will simplify the expansion of the framework and increase the compatibility of the new MMUs with the existing ones. It has to be noted that this is still an open topic under research and final solutions will only be available once the number of MMUs and use cases covered are large enough to cover the most distinct scenarios.

3.5. MMU description file (manifest)

One of the key aspects of MMU management is the MMU description file that is provided in the JSON format [27]. This file contains both general information about the MMU provided for the end-user, as well as implementation information that allows the computer code to recognize the MMU programming language, the motion type that the MMU performs, or an ID that is used to uniquely identify MMUs in library synchronization events.

Currently, a tool for MMU description editing is under preparation and will be a part of the framework released as an open-source on the MOSIM GitHub pages [28] and therefore, the manual description file preparation will be addressed in this section. It has to be noted that the JSON format is sensitive to string delimiter presence around numeric values. When delimiters are added, the number is treated as a string and can raise JSON parsing errors on MMU startup or management in library. The decimal separator for floating-point numbers in the JSON format is a dot. The thousands separator or spaces in numeric values are not allowed. In strings, the backslash character has to be escaped with another backslash character. Similarly, any quotation mark that is the same as the opening and closing quotation symbol within a string needs to be escaped with a backslash. The JSON allows the use of either double quotes or single quotes as quotation marks to delimit strings. If a double quotation is used to identify a string, then single quotations contained within such a string do not have to be escaped. Analogically, when single quotations are used to delimit the string, double quotation inside can be used without escaping. Fields Name, ID, AssemblyName, Language and MotionType are the most important parameters and are required for valid MMU description. The Name indicates human readable name of the MMU; it is not used directly in the framework, so it is not sensitive to the execution of the MMU, but has a value for the MMU library management. The ID is a string that should be unique at least within the MMU library where the MMU is used.

To maintain uniqueness of IDs, it is suggested to construct an ID using the author's e-mail or the vendor's internet domain as the first part, separated by a colon with the MMU name and a slash with the MMU version. This way, while Internet domains are globally unique, only MMU names have to be unique within the organizations owning a domain. In case an organization has multiple independent MMU developers and does not keep track of the MMU IDs used within its structures, the author's e-mail is a safe choice for the ID prefix. The same applies to e-mail accounts of individual developers that are hosted on one of the popular e-mail provider's domains that contain millions of users, then, instead of the domain, the author's e-mail is a safe choice. The ID cannot contain trailing and preceding spaces, but can contain spaces inside. The reason is that, while processing, preceding and trailing spaces and other white characters are stripped. The AssemblyName field has to contain an MMU executable file name, including the extension or dynamic link library name of the MMU that contains the entry point.

3.6. Running MMUs with the Launcher

Once the MMU is compiled, it can be placed within a local MMU library, becoming a part of the Launcher software. If only the MMU is implemented in one of the programming languages for which the adapter is available, the MMU will be loaded and can be used by the framework. Setting up the basic framework tools and the launcher is presented in the documentation on the GitHub (https://github.com/Daimler/MOSIM_Core/wiki/MMI-Framework-Setup-Guide).

3.7. Testing MMUs in example scene

If the MMU implements the motion type that is already available in the task editor and the breakdown is ready in the reasoning engine, then it is sufficient to disable other MMUs that implement the same motion type or to set up the new MMU as the first one on the MMU priority list within its motion type. Then, setting a new task in the task editor and running the simulation will result in using the new MMU as long as it does not fail during initialization.

In case MMU implements a new motion type or a motion type for which its behavior is not available, a user can either define a new breakdown, insert it into the reasoning engine library and define a new task in the task editor to try out the MMU. A direct MMU call can also be made from the target engine to run just the specific MMU with the user script.

3.8. MMU distribution package

The MMU can be distributed to other users using a zip archive. The structure of the zip archive content has been standardized in the MOSIM project and is presented in Fig. 1. An important part of the structure is the description.json file

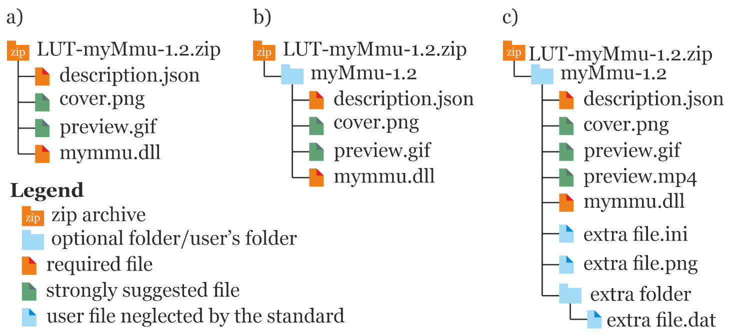

Fig. 1. MMU folder structure, a) simple structure without root folder, b) structure with root folder, c) structure with root folder, optional files, and user files

that contains the MMU manifest presented in earlier sections. This manifest is used by the MMU library and the launcher to instantiate the MMU, classify its function and connect it to the related behavior. The manifest is also used to track MMU updates and implement automatic update checking.

The manifest file and the zip archive with MMU can be created using the MOSIM tool called the MMUDescriptionEditor, which performs a sanity check for all the parameters and utilizes information about framework-supported parameters and their values directly from the api.mosim.eu guaranteeing parameters that are up-to-date with the framework development.

The second most important file in the MMU distribution package is the dll containing the actual MMU. The description file has to explicitly indicate which dll file in the archive is the MMU implementation. Additionally, files containing images for web shop and library management usage can be included, as well as a gif animation (to be used as an animated thumbnail) and an mp4 video showing more details about the MMU. The intended use of the media files is presentation of the MMU in web shops as well as in MMU libraries. The MMU folder structure can be as simple as a zip archive containing the only two required files, namely the description.json and the actual MMU dll. However, to ease manual MMU management, a root folder can be added, as shown in Fig. 1.

There is no limit on the number of files and folders accompanying MMUs or their types, however, the files described above have to follow naming standards and their content must obey content restrictions defined by the MMU library standard to be correctly recognized by the framework and be validated by MOSIM framework tools. An online MMU validation tool is currently under development.

4. Example MMU implementation

This section presents two MMU implementations that show how a C# project can be created and compiled with a simple nod and shake MMU, and how pre-recorded motion data can be used in a purely motion-driven MMU implemented in the Unity C#. The second MMU performs a reaching action using the target object's location to determine the reach point. Both MMUs show how run-time and how they can be used in the MMU, how the scene can be accessed from the MMU, how to access services, and how to implement event creation during MMU execution. In addition, the second MMU example also shows how motion blending functionality in the Unity can be used, and presents accessing scene parameters and their interpretation. Finally, this section presents the testing procedure of the MMU and shows how it can be used with the task editor and the reasoning engine. Complete MMU codes are available on LUT's GitHub repository.

4.1. Simple nod and shake head MMU implementation in C#

Nodding and shaking the head from left to right are common ways of communicating non-verbally, indicating a yes or no response. While Bulgaria utilizes the opposite logic for these responses compared to most of the countries, it is assumed in the implementation that, by default, nodding means a yes and shaking the head from left to right means a no. To handle the Bulgarian understanding of the gesture, the `inverseLogic` persistent parameter is implemented such that it can be passed onto the MMU during startup or using a special configuration call.

The used MMU will need one runtime parameter called the Response, that will accept a Boolean data type with binary values of 0 or 1, where 0 is interpreted as false, and 1 as true. If the `inverseLogic` parameter is not specified, the default value will be false, meaning that the Response parameter of value 1 will result in nodding the head, and the Response parameter with value 0 will result in shaking the head. The MMU will support only the basic events: `start`, `ready`, `end`, and `abort`. The MMU will implement basic procedure calls: `CheckPrerequsites`, `DoStep`, `Abort`, `Setup`, and the generic method `Consume` that will be used to pass the general persistent MMU configuration during the runtime, and those parameters will be restored on reload.

The MMU will only actuate the neck joint and will not affect the avatar pose in any other way. This means that the most natural-looking response will happen for the avatar during standing and sitting positions. It also implies that if the avatar would be in a bent position, the avatar's head will not be raised before shaking for a no response. Nevertheless, for demonstration purposes, this assumption should be sufficient.

The motion data for nodding and shaking is based on the predefined neck joint motion data points: 0, 45, -45, 0 for shaking, and 0, 10, -30, 0 for nodding. The data points are expressed in degrees and represent a single neck rotation angle. On the `DoStep` method, linear interpolation will be used to provide output neck angles for any valid time step. If the initial neck joint angle does not represent a straight neck, the first motion frames will interpolate turning the head from the initial pose to the default nodding/shaking starting pose, consequently the overall time for the motion will be longer. This demonstrates that MMUs can simulate more realistically action timing, as they take into account transition states between individual motions as compared to a pure animation approach, where transitions can happen in an abrupt way and animation time is fixed to the duration of the clip.

It has to be noted that if the time step is larger than the overall duration of the motion, the pose returned through the `DoStep` function call will represent the initial nodding/shaking pose. In most cases, a time step within 0.01 to 0.1 second will provide a smooth animation. Smaller time steps lead to smoother animations, but if the number of frames per second that the computer can generate is exceeded, then the animation might be lagging behind the real-time or intermediate steps might not be visualized.

4.1.1. Code structure

MMU class is derived from the `MMUBase` class. The following methods are implemented by overriding base class methods: `initialize`, `AssignInstruction`, and `DoStep`. In addition, the method for interpolating rotations between two given poses is provided as the `FromToRotation` function. The MMU is contained in its own namespace. The MMU utilizes the functionality of the `MMICSharp`, `MMIStandard`, and System libraries.

```
using MMICSharp.Common;
using MMICSharp.Common.Attributes;
using MMIStandard;
using System;
using System.Collections.Generic;

namespace NodAndShakeMMU {
    public class NodeAndShakeMMUImpl : MMUBase {
     public override MBoolResponse Initialize(MAvatarDescription
         avatarDescription, Dictionary<string, string> properties) {
            base.Initialize(avatarDescription, properties);
            //Setup the skeleton access
            this.SkeletonAccess = new IntermediateSkeleton();
            this.SkeletonAccess.InitializeAnthropometry(
                avatarDescription
            );

            //Get initial rotations
            this.initialHeadRotation =
                SkeletonAccess.GetLocalJointRotation(
                AvatarDescription.AvatarID, MJointType.HeadJoint
            );
            this.initialNeckRotation =
                SkeletonAccess.GetLocalJointRotation(
                AvatarDescription.AvatarID, MJointType.C4C5Joint
            );
            //return success response of the initialization
            return new MBoolResponse(true);
        }

    public override MBoolResponse AssignInstruction(MInstruction
        instruction, MSimulationState simulationState) {
            base.AssignInstruction(instruction, simulationState);
            //Here instruction processing logic needs to be inserted
            //return success response
            return new MBoolResponse(true);
        }

    public override MSimulationResult DoStep(double time,
        MSimulationState simulationState) {
            //Create a new simulation result
            MSimulationResult result = new MSimulationResult() {
                Events = simulationState.Events ?? new
                    List<MSimulationEvent>(),
                Constraints = simulationState.Constraints ?? new
                    List<MConstraint>(),
                SceneManipulations = simulationState.SceneManipulations??
                    new List<MSceneManipulation>(),
                Posture = simulationState.Current
            };
        //here logic for performing single animation step needs to be
            inserted
        return result;
        }

    private static MQuaternion FromToRotation(MVector3 from, MVector3 to)
        {
            //Normalize both input vectors
            from = from.Normalize();
            to = to.Normalize();

            //Estimate the rotation axis
            MVector3 axis = MVector3Extensions.Cross(from,
                to).Normalize();

            //Compute the phi rotation angle
            double phi = Math.Acos(MVector3Extensions.Dot(from, to)) /
                (from.Magnitude() * to.Magnitude());

            //Create a new quaternion representing the rotation
            MQuaternion result = new MQuaternion() {
                X = Math.Sin(phi / 2) * axis.X,
                Y = Math.Sin(phi / 2) * axis.Y,
                Z = Math.Sin(phi / 2) * axis.Z,
                W = Math.Cos(phi / 2)
            };

            //Perform not a number check and return identity quaternion
                if quaternion components are invalid
            if (double.IsNaN(result.W) || double.IsNaN(result.X) ||
                double.IsNaN(result.Y) || double.IsNaN(result.Z))
                result = new MQuaternion(0, 0, 0, 1);

            //Return the estimated rotation
            return result;
        }
    }
}
```

MOSIM data types used in the code snippet above are defined as follows: the `MBoolResponse` is a structure that contains the required Boolean value indicating success or failure of the instruction as well as an optional list of strings for log data, that is intended to transmit error and warning messages as addition to the Boolean response. It can also be used to log successful events if the MMU developer so desires. A Thrift definition of the `MBoolResponse` is presented below. The Thrift interface definition is limited to four fields per entry: id - used to identify parameter, keyword `required` or `optional` indicating whether parameter can be omitted or not; type filed, and finally parameter name. This is translated by the Apache Thrift compiler to any supported programming language that the user wants to utilize for implementation.

```
struct MBoolResponse {
 1:required bool Successful;
 2:optional list<string> LogData;
}
```

The `MAvatarDescription` defines the avatar pose, properties and the avatar ID, which allows avatar identification in multi-avatar simulation scenarios. The Thrift definition of `MAvatarDescription` is as follows:

```
struct MAvatarDescription {
 1: required string AvatarID;
 2: required MAvatarPosture ZeroPosture;
 3: optional map<string,string> Properties;
}
```

Processing of runtime parameters is done in the `AssignInstruction` method. The first parameter of the `AssignInstruction` is of `MInstruction` type, which is defined in the Thrift as follows:

```
struct MInstruction {
 1: required string ID;
 2: required string Name;
 3: required string MotionType;
 4: optional map<string,string> Properties;
 5: optional list<constraints.MConstraint> Constraints;
 6: optional string StartCondition;
 7: optional string EndCondition;
 8: optional string Action;
 9: optional list<MInstruction> Instructions;
}
```

The `ID` instruction is assigned to maintain control over start and end conditions of individual instructions and appears as a part of event descriptions raised by MMUs. For example, if the walk instruction of the walk MMU has an `ID` of "WalkMMU-1", then if walk should be followed by reach, the reach MMU should be instantiated with the `MInstruction` field `StartCondition` set to "WalkMMU-1:end".

Construction of the condition is the MMU ID followed by a colon and the event name. Despite the `StartCondition` or `EndConditions` being strings, to maintain compatibility with the future framework versions, their values should be set using framework defined constants. For example, the start condition after the end of the previous MMU should be set this way:

```
    StartCondition = walkInstruction.ID + ":" +
        mmiConstants.MSimulationEvent_End,
```

where `mmiConstraints.MSimulationEvent_End` defines the string representing the end event; if it follows an MMU that does not have a fixed end as it cycles through specific motion (for example idle MMU), then it would be more appropriate to use the `mmiConstraints.MSimulationEvent_CycleEnd`, as this event should be raised at the end of each animation cycle. So, the second MMU would follow the idle one after one cycle is executed.

Each instruction must have a unique `ID` assigned to it. To generate such an ID, the `MInstructionFactory.GenerateID` function should be used, which guarantees uniqueness of the ID within a runtime environment. A `Name` parameter has to be set to the MMU name, and a `MotionType` to the motion type represented by the specific MMU. These two parameters are matched by MMU adapter to the specific MMU and then such instruction is forwarded to the MMU. Therefore, `Name` and `MotionType` parameters do not have to be processed by the MMU, they are intended for MMU libraries and the co-simulator for MMU selection and organization. The Properties dictionary is intended for passing runtime parameters to the MMU in the `AssignInstruction` method. The MMU should parse the `Properties` value of the instruction parameter in search of understandable parameters. The parameters that are not required should be ignored.

As all parameters and their names are passed as strings, the list of parameters is easily extendable. The dictionary structure imposes that the order of parameters is irrelevant. MMU developers are encouraged to reuse parameter names and predefined values utilized already by other MMUs. To increase cross-compatibility as much as possible, the same parameter lists should be used for MMUs representing the same motion type. To get a list of already supported parameters and their values api.mosim.eu endpoint should be used, before defining new parameters. To keep the parameter space consistent and avoid any character set translation problems, parameter names and values should be written in English using Latin characters and numbers only. The api.mosim.eu page will be extended in the near future to support upload of MMU description files and to maintain up-to-date lists of parameters and available MMUs on the market. In addition to the dictionary of parameters, more complex parameters can be given as constraints. As described earlier, the constraints mostly apply to geometrical values like positions, rotations or ranges that allow defining areas and volumes as well as the avatar's joint limits or poses. Finally, the Instructions List can be provided to the MMU as a set of other instructions that should be processed in the order they are specified within a single `AssignInstruction` call.

The proposed nod and shake MMU processes just one runtime parameter "Response" that takes 0 or 1 value. Processing of this parameter boils down just to two one if condition in the `AssignInstruction` method:

```
if (instruction.Properties.ContainsKey("Response"))
this.Response = (instruction.Properties["Response"]=="1")

if (this.Response) {
//nod motion
trajectory.Add(new MTransform("",new MVector3(0,0,0),new
    MQuaternion(0,0,1,0)));
trajectory.Add(new MTransform("",new MVector3(0,0,0),new
    MQuaternion(0,0,1,nodMax)));
trajectory.Add(new MTransform("",new MVector3(0,0,0),new
    MQuaternion(0,0,1,nodMin)));
trajectory.Add(new MTransform("",new MVector3(0,0,0),new
    MQuaternion(0,0,1,0)));
}
else {
//shake motion
trajectory.Add(new MTransform("",new MVector3(0,0,0),new
    MQuaternion(0,1,0,0)));
for (byte i=0; i<2; i++) {
 trajectory.Add(new MTransform("",new MVector3(0,0,0),new
     MQuaternion(0,1,0,-shakeLimit)));
 trajectory.Add(new MTransform("",new MVector3(0,0,0),new
     MQuaternion(0,1,0,shakeLimit)));
 trajectory.Add(new MTransform("",new MVector3(0,0,0),new
     MQuaternion(0,1,0,0)));
 }
}
```

Firstly, it is checking if the parameter has been defined, and secondly, if it is set to 1 or not. This implies that the Response parameter has a default value if it is not specified. The default value is defined in `this.response` field declaration. The value of "1" will be interpreted as Boolean true, and any other value as false.

Knowing which response is requested, motion trajectory planning is performed by specifying key points for the nod motion. The parameters for the `trajectory.Add` method are `ID`, translation, and rotation defining joint pose. The `ID` parameter is left blank as it is not needed in this context, the translation is set to zero in every case as the neck joint is modeled as a spherical joint, and the rotations define steps from the look ahead position, through reaching max and min limits to look ahead. For shaking, two shakes are defined within a loop. The number of head shaking could also be provided as a parameter, then loop limits have to be adjusted accordingly.

Finally, in the `DoStep` method, a logic for outputting neck state at the end of every simulation step needs to be added. Depending on the runtime `Response` parameter, either nod or shake action is simulated. In this example, the values will be interpolated within limits defined as constants: `shakeLimit`, `nodMax` and `nodMin`. As shaking is symmetrical, only one angular limit is defined as an absolute angle value. For nodding, Max and Min values are defined, allowing the setting of different limits. A value of 0 is used as the middle point, therefore it does not need to be defined explicitly as a constant.

```
public class NodeAndShakeMMUImpl : MMUBase {
        const float shakeLimit = 45;
        const float nodMax = 10;
        const float nodMin = -30;
    byte trajectoryIndex = 0 ;
    List<MTransform> trajectory = new List<MTransform>();
    ...
 }
```

The global constants are defined at the beginning of the class. Additionally, the `trajectoryIndex` variable is defined to track the current target point for the motion. Then, the `DoStep` method can be completed by the neck movement code. The first neck rotation has to be extracted from the avatar object using the `SkeletonAccess`.

```
MVector3 currentNeckPosition = this.SkeletonAccess.GetGlobalJointRotation(
    this.AvatarDescription.AvatarID,MJointType.HeadJoint
);
MQuaternion currentNeckRotation =
    this.SkeletonAccess.GetGlobalJointRotation(
    this.AvatarDescription.AvatarID,MJointType.HeadJoint
);
```

Then, a new joint angle has to be computed based on the time increment since the last time step. In most cases, the time step can be expected to be fixed, but to handle uneven time steps, the actual interval must be used.

```
nextPose = this.DoLocalMotionPlanning(currentVelocity,
    this.angularVelocity, TimeSpan.FromSeconds(time),
    currentNeckPosition, currentNeckRotation, currentNeckPosition,
    nextNeckRotation);
 //Check if close to current target -> move to next target -> To do
     consider rotation
    if (
        (nextPose.Position.Subtract( trajectory[trajectoryIndex].Position
            )).Magnitude() < this.translationThreshold  &&
            MQuaternionExtensions.Angle(nextPose.Rotation,
            trajectory[trajectoryIndex].Rotation)< this.rotationThreshold
            && trajectoryIndex < trajectory.Count - 1
        ) {
        trajectoryIndex++;
        //end of animation frames
        if (trejectoryIndex>=trajectory.Count)
        result.Events.Add(new MSimulationEvent(this.instruction.Name,
            mmiConstants.MSimulationEvent_End, this.instruction.ID));
    }
 this.SkeletonAccess.SetGlobalJointRotation(
     simulationState.Current.AvatarID, MJointType.C4C5Joint, nextPose);
```

The function `DoLocalMotionPlaning` interpolates values between animation keyframes. Whenever the keyframe is reached within the tolerance limit, the next keyframe is used for motion planning. In any case, the `SkeletonAccess` is used to set a new joint state using the `SetGlobalJointRotation` method. If the last animation keyframe is reached, the End event is raised, indicating that the MMU can be terminated by the co-simulator.

4.2. Reach MMU in Unity C# based on motion blending

Motion blending is a technique widely used with 3D animations, especially generated by motion capture or key-framing. Originally, this technique is used for creating smooth transitions between two homogeneous motions, e.g., running and walking. Nevertheless, there are no strict limitations for the homogeneity of blended motions. Therefore, it may be possible to apply a wide variety of motion types to generate a new set of motions.

During the motion blending, two or more motions are merged with a certain ratio called the motion weight. The procedure may vary depending on the game engine chosen. In the example of the Unity game engine, motion blending weights could be computed with the blend trees [29], which provide five blending types: 1D, 2D simple directional, 2D freeform directional, 2D freeform Cartesian and direct. Among the listed types for this specific application, a suitable type is the 2D freeform directional motion blending, which allows for choosing the motion blending weights on a plane. The space of the motion blending can be observed in Fig. 2. In the figure, dots represent single motion primitives. By combining specific motions with blending weights, it is possible to reach any place in the space, including the points behind the avatar.

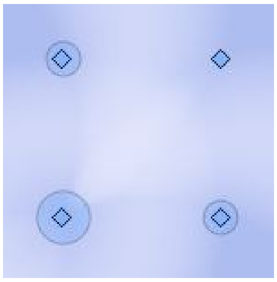

Fig. 2. Motion blending space

4.2.1. Motion selection

The choice of motions must be thorough, because it will affect further blending. It is essential to use only motions related to the motion groups, such as reaching. It is important to avoid choosing different motion groups, for example, picking up an object from the ground or making an air squat. Using homogeneous motions will provide smooth and realistic motion blending. Experiments show that for creation of the reach motion at least six reaching motions should be used describing reaching in the following directions: front-down, front-middle, front-up, side-down, side-middle, and side-up. It is sufficient to describe such motions for just one arm, as the other arm positions can be obtained from mirroring the single-side motion.

Knowing the position of the arm at the end of the motion and the corresponding blending weights configuration, it is possible to reach an object within some volume around the avatar. Reach positions are directly linked to the blending weights, and saving a table of blending weights and corresponding reach positions for a large number of positions allows one to define good reach accuracy. In this example, two blending weights are defined and a normalized space of blending weights ranging from zero to one with 0.1 increments is saved. This space is presented in Fig. 2. The resulting table contains 10000 reach positions and the associated blending weights. The average space between the points is 30 millimeters, while the maximum does not exceed 40 millimeters. Those values are sufficient for the avatar animation to reproduce decent reaching motions in the work space. If higher accuracy is required, then interpolation of the blending weights between several points closest to the desired reach point can be used, or a data table with smaller increments between weights can be generated. Finding the weight for reaching a specific point is determined through a table search for the closest point coordinates and by using the associated weights. The resulting lookup table is saved to a CSV file [30]. It is important to mention that reach coordinates must be relative to the avatar's position to allow for reaching no matter where the avatar is located within the scene, but not to the values in the global frame of reference which would allow only reaching target points for an avatar placed in a specific position. In this example, the avatar's pelvis coordinates are used as a reference for the target reach points positions.

4.2.2. MMU generation in Unity

MMU can be generated in the Unity using the `MMU Generation script` provided as a part of the MOSIM open-source framework. The use of the script requires completing simple setup steps: creating of an empty Unity project, importing the MMU Generation script package, inserting an avatar and giving it the same name as the final MMU name should be. The script allows one to specify information that will be placed in the description manifest file using the graphical user interface. Currently, only basic parameters are supported through the script, so additional parameters need to be added using the text editor. The script uses the scene information to prepare the MMU template script attached to the avatar. The avatar also contains an animator script that can be used to supply motion capture data that the avatar should use. Finally, the user needs to modify the template MMU script attached to the avatar to process input parameters and therefore modify the avatar's animators behavior. This method is simpler than generating an MMU using the C# code directly, as it provides visual feedback and more debugging capabilities.

```
using MMICSharp.Common;
using MMIStandard;
using MMIUnity;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class BlendMMU : UnityMMUBase {
    public Dictionary<string,MJointType> BoneTypeMapping = new Dictionary<string,MJointType>();

    private Animator animator;

    private MInstruction instruction;

    string goal_object_name;

    Vector3 point_position;
    Vector3 goal_position;
    Transform pelvis_transform;

    // Distance to the point
    float distance;

    // Lowest distance so far
    float lowest_distance = 10;

    // Number of the lowest point
    int point_number;

    // Reachable points names array
    //string[] points = { "Sphere", "Sphere1", "Sphere2", "Sphere3" };

    // Reachable points positions relative to pelvis
    float[] points_x = { 0.2f, 0.2f, 0.1f, 0.1f };
    float[] points_y = { 0.1f, -0.1f, -0.1f, 0.1f };
    float[] points_z = { 0.8f, 0.8f, 0.9f, 0.9f };

    // Arrays of corresponding to names blending ratios
    float[] blend_x_array = { 0.5f, 0.5f, 0.5f, 0.75f };
    float[] blend_y_array = { 0.25f, 0.15f, 0.5f, 0.25f };

    protected override void Awake() {
        //Assign the name of the MMU
        this.Name = "BlendMMU";

        //Assign the motion type of the MMU
        this.MotionType = "Pose/Reach";

        //Get the animator
        this.animator = this.GetComponent<Animator>();

        //Disable the animator at the beginning
        this.animator.enabled = false;

                //It is important that the bone assignment is done before the base class awake is called
                base.Awake();
    }

    // Start is called before the first frame update
    protected override void Start() {
        base.Start();
    }

    public override MBoolResponse Initialize(MAvatarDescription
        avatarDescription, Dictionary<string, string> properties) {
        //Execute the instruction on the main thread
        this.ExecuteOnMainThread(() =>
        {
            //Call the base class initialization
            base.Initialize(avatarDescription, properties);

            //Set culling mode to always animate
            this.animator.cullingMode = AnimatorCullingMode.AlwaysAnimate;

            //Deactivate the animator we want to trigger it manually in
                the dostep
            this.animator.enabled = false;
        });

        goal_object_name = properties["GoalObjectName"];

        //Return success
        return new MBoolResponse(true);
    }
```

Before motion blending, the MMU calculates the distance between the goal and all available points in the dataset. All the positions are converted to positions that are relative to the avatar transform. If the goal to the point exceeds 0.1 meters, the MMU reports an error.

```
    public override MBoolResponse AssignInstruction(MInstruction
        instruction, MSimulationState state) {
        //Assign the instruction to the class variable
        this.instruction = instruction;

        //Execute the instruction on the main thread
        this.ExecuteOnMainThread(() =>
        {
            //Assign the posture
            this.AssignPostureValues(state.Current);

        });

        // Find pelvis transform
        pelvis_transform = GameObject.Find("pelvis").transform;

        // Find goal obect and get transform
        goal_position =
            GameObject.Find(goal_object_name).transform.position;
        Vector3 goal_relative_position =
            getRelativePosition(pelvis_transform, goal_position);

        for (int i = 0; i <= points_x.Length - 1; i++) {
            point_position.Set(points_x[i], points_y[i], points_z[i]);
            Vector3 point_relative_position =
                getRelativePosition(pelvis_transform, point_position);

            distance = Vector3.Distance(point_relative_position,
                goal_relative_position);
            if (distance < lowest_distance) {
                lowest_distance = distance;
                point_number = i;
            }
        }

        Debug.Log("Lowest distance: " + lowest_distance + " Corresponding
            blending values: " + "X " + blend_x_array[point_number] + " "
            + "Y " + blend_y_array[point_number]);

        if (lowest_distance < 0.1) {
            this.animator.SetFloat("x", blend_x_array[point_number]);
            this.animator.SetFloat("y", blend_y_array[point_number]);
        }
        else {
            Debug.Log("Target is not within reach!");
        }
        return new MBoolResponse(true);
    }

    public override MSimulationResult DoStep(double time,
        MSimulationState state) {
        this.animator.enabled = true;

        //Create a new simulation result
        MSimulationResult result = new MSimulationResult() {
            Posture = state.Current,
            Constraints = state.Constraints!=null ? state.Constraints:
                new List<MConstraint>(),
            Events = state.Events !=null? state.Events: new
                List<MSimulationEvent>(),
            SceneManipulations = state.SceneManipulations!=null ?
                state.SceneManipulations: new List<MSceneManipulation>(),
        };

        //Execute the instruction on the main thread (required in order
            to access unity functionality)
        this.ExecuteOnMainThread(() =>
        {
            //Update the animator
            this.animator.Update((float)time);

            //Get the current posture of the after in the intermediate
                skeleton representation
            result.Posture = this.GetRetargetedPosture();

            //To do -> Process the events and return it as result
        });

        mmiConstants.MSimulationEvent_End, this.instruction.ID));
        return result;
    }

    public override List<MConstraint> GetBoundaryConstraints(MInstruction
        motionCommand) {
        return new List<MConstraint>();
    }

    public override MBoolResponse CheckPrerequisites(MInstruction
        instruction) {
        return new MBoolResponse(true);
    }
```

The method below converts the global position to the relative position based on the transform of the avatar.

```
    public static Vector3 getRelativePosition(Transform origin, Vector3
        position) {
        Vector3 distance = position - origin.position;
        Vector3 relativePosition = Vector3.zero;
        relativePosition.x = Vector3.Dot(distance,
            origin.right.normalized);
        relativePosition.y = Vector3.Dot(distance, origin.up.normalized);
        relativePosition.z = Vector3.Dot(distance,
            origin.forward.normalized);

        return relativePosition;
    }
}
```

5. Conclusion

Simulation of manual assembly work is of core importance for large manufacturing companies, where manual labor is still important from the product quality point of view standpoint and flexibility that human workers offer as compared to robotic stations. MOSIM framework aims at simplifying the steps required to perform such simulation and turn complex animation-based modeling into reasoning of task lists that are common to the simulation framework and actual factory workers.

Basic motions utilized for the assembly of complex human motions are in the MOSIM framework bundled to the MMUs together with the logic for object handling and environment interactions. This allows logic encapsulation, scene modification capabilities, other avatar awareness, and motion planning into self-contained software pieces that are modular and interchangeable. The motion type concept that has been introduced into the MMU description linked with an online service used to track and organize new motion types and other parameters makes the framework open for new applications while at the same time keeping semantic information about motion data contained in an MMU. The two basic MMU examples presented in this paper should help new developers to start using the framework and building new MMUs based on either motion blending or direct joint manipulation approaches that can rely on any logic or motion synthesis algorithm.

In the near future, MMUs and behavior models will be provided on the MOSIM project web site. In addition, a shopping platform offering MMUs is under development to ease the exchange of MMUs between providers and consumers. The MMI standard is currently supported by major automotive manufacturers, industry automation providers, IT, and simulation technology providers.

Acknowledgements

This work is a part of the international ITEA3 MOSIM project that is funded in each country by a designated funding agency. The Authors of this publication are affiliated with the Lappeenranta University of Technology that has been funded by Business Finland (funding decision number: E!7632).

References

[1] Manufacturing statistics – NACE Rev. 2. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Manufacturing_statistics_-_NACE_Rev._2&oldid=502915. 2021-03-04.

[2] D. Sabadka, V. Molnar, and G. Fedorko. Shortening of life cycle and complexity impact on the automotive industry. *TEM Journal*, 8(4):1295–1301, 2019. doi: 10.18421/TEM84-27.

[3] J. Ajaefobi and R. Weston. Modelling human systems in support of process engineering. In G. Zülch, H. Jagdev, and P. Stock, editors, *Integrating Human Aspects in Production Management. IFIP International Conference for Information Processing*, volume 160, pages 3–16, Boston, MA, 2005. Springer US. doi: 10.1007/0-387-23078-5_1.

[4] D. Lämkull, L. Hanson, and R. Örtengren. A comparative study of digital human modelling simulation results and their outcomes in reality: A case study within manual assembly of automobiles. *International Journal of Industrial Ergonomics*, 39(2):428–441, 2009. doi: 10.1016/j.ergon.2008.10.005.

[5] D. B. Chaffin, C. Nelson, et al. *Digital Human Modeling for Vehicle and Workplace Design*. Society of Automotive Engineers Warrendale, PA, USA, 2001.

[6] H.O. Demirel and V.G. Duffy. Applications of digital human modeling in industry. In V.G. Duffy, editor, *Digital Human Modeling*, pages 824–832. Springer, 2007. doi: 10.1007/978-3-540-73321-8_93.

[7] A. Naumann and M. Rötting. Digital human modeling for design and evaluation of human-machine systems. *MMI Interaktiv*, 12:27–35, 2007.

[8] V. Duffy. *Handbook of Digital Human Modelling: Research for Applied ergonomics and Human Factors Engineering*. CRC Press, 2008. doi: 10.1201/9781420063523.

[9] C.B. Phillips and N.I. Badler. JACK: A toolkit for manipulating articulated figures. In *Proceedings of the 1st Annual ACM SIGGRAPH Symposium on User Interface Software*, UIST '88, page 221–229, New York, NY, USA, 1988. doi: 10.1145/62402.62436.

[10] R. Gilbert, R. Carrier, J. Schiettekatte, C. Fortin, B. Dechamplain, H.N. Cheng, A. Savard, C. Benoit, and M. Lachapelle. SAFEWORK: Software to analyse and design workplaces. In G.R. McMillan et al., editors, *Applications of Human Performance Models to System Design*, pages 389–396. Springer, 1989. doi: 10.1007/978-1-4757-9244-7_28.

[11] K. Abdel-Malek, J. Yang, J. H. Kim, T. Marler, S. Beck, C. Swan, L. Frey-Law, A. Mathai, C. Murphy, S. Rahmatallah, and J. Arora. Development of the virtual-human santostm. In V.G. Duffy, editor, *Digital Human Modeling*, pages 490–499. Springer, 2007. doi: 10.1007/978-3-540-73321-8_57.

[12] MSC Adams software. https://www.mscsoftware.com/product/adams. Accessed: 2021-03-22.

[13] S.L. Delp, F.C. Anderson, A.S. Arnold, P. Loan, A. Habib, C. T. John, E. Guendelman, and D. G. Thelen. OpenSim: open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, 54(11):1940–1950, 2007. doi: 10.1109/TBME.2007.901024.

[14] P.-B. Wieber, F. Billet, L. Boissieux, and R. Pissard-Gibollet. The humans toolbox, a homogenous framework for motion capture, analysis and simulation. In *International Symposium on the 3D Analysis of Human Movement*, 2006.

[15] H.-J. Wirsching. Human solutions RAMSIS. In S. Scataglini and G. Paul, editors, *DHM and Posturography*, pages 49–55.Academic Press,2019.doi:10.1016/B978-0-12-816713-7.00004-0.

[16] M. Hovanec, P. Korba, and M. Solc. TECNOMATIX for successful application in the area of simulation manufacturing and ergonomics. In *15th International SGEM Geoconference on Informatics*, Albena, Bulgaria, 2015.

[17] J. Rasmussen. The AnyBody modeling system. In S. Scataglini and G. Paul, editors, *DHM and Posturography*, pages 85–96. Academic Press, 2019. doi: 10.1016/B978-0-12-816713-7.00008-8.

[18] Simcenter Madymo. https://www.plm.automation.siemens.com/global/en/products/simcenter/madymo.html. Accessed: 2022-04-07.

[19] 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. *Procedia CIRP*, 72:768–773, 2018. doi: 10.1016/j.procir.2018.03.281.

[20] FMI standard. https://fmi-standard.org/. Accessed: 2021-03-22.

[21] Unity. https://unity.com/. Accessed: 2021-03-22.

[22] Unreal engine. https://www.unrealengine.com/en-US/. Accessed: 2021-03-22.

[23] F. Gaisbauer, E. Lampen, P. Agethen, and E. Rukzio. Combining heterogeneous digital human simulations: presenting a novel co-simulation approach for incorporating different character animation technologies. *The Visual Computer*, 37:717–734, 2020. doi: 10.1007/s00371-020-01792-x.

[24] MOSIM. https://mosim.eu. Accessed: 2021-03-22.

[25] A. Safonova, J.K. Hodgins, and N.S. Pollard. Synthesizing physically realistic human motion in low-dimensional, behavior-specific spaces. *ACM Transactions on Graphics*, 23(3):514–521, 2004. doi: 10.1145/1186562.1015754.

[26] K. Rakowski. *Learning Apache Thrift*. Packt Publishing Ltd, 2015.

[27] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström. JSON-LD 1.0. *W3C recommendation*, 16:41, 2014.

[28] MOSIM download section. https://mosim.eu/download.php. Accessed: 2021-03-22.

[29] K. Pietroszek, P. Pham, S. Rose, L. Tahai, I. Humer, and C. Eckhardt. Real-time avatar animation synthesis from coarse motion input. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology*, Gothenburg, Sweden, 2017. doi: 10.1145/3139131.3141223.

[30] RFC CSV file specification. https://tools.ietf.org/html/rfc4180. Accessed: 2021-03-29.