AIAA SciTech Forum
January 3-7, 2022, San Diego, CA & Virtual
AIAA SCITECH 2022 Forum

10.2514/6.2022-0808

Extending Simulink with Blocks Managing Complex Objects Collaboratively – Application to Air-to-Air Refueling Simulation

Nicolas Fezans* and André Koloschin†
DLR (German Aerospace Center), Institute of Flight Systems, Braunschweig, Germany

The present paper describes a Simulink® extension that allows creating, managing, and exploiting complex object-oriented structures by adding several s-function blocks to the model and by configuring them. The paper describes first the simulation mechanism in a generic way and then shows a concrete application. This application originates from the authors' research on automation of air-to-air refueling maneuvers and has motivated this development. In this application, the developed Simulink® extension is used to manage reference frames, perform kinematic computations, modeling complex wind fields, and perform low-order aerodynamics computations based on these modeled wind fields. From a Simulink® user/modeler perspective, the whole add-on consists of a series of Simulink® blocks with masks to configure them, that can be placed at the most convenient locations in the block diagram and permit to develop and use advanced features within a few minutes, which would otherwise have required a much larger modeling effort and that would have been error-prone.

doi: 10.2514/6.2022-0808

I. Introduction

MATLAB®/Simulink® is currently one of the most popular simulation software packages available for control theory, flight simulation and many other applications. Many additional toolboxes can be bought/added to extend its functionalities and add other paradigms. At its core[§], Simulink® is a *signal-based* simulation environment with a graphical (block diagram) editor. Users program the dynamic equations of the system that they are modeling and simulating by connecting blocks with each other. It provides a fairly large library of blocks and users can extend it by programming their own blocks; this possibility and how to push it one step further is at the core of the present work.

One of the other main, and increasingly popular, approaches for building simulation models is the so-called *component-based* approach. The signal-based simulation paradigm is rather focused on the information that is passed from block to block, which, equivalently, can be seen as a description of the system differential equations and how they are inter-connected. Among various other alternatives, the *component-based* simulation paradigm focuses instead on the physical connections between the system components. Once the components have been connected to each other and some component level dynamic model and properties have been set, the equations of the overall system can be produced and used for simulation. Users of *component-based* modeling software packages usually do not interact with the overall system equations but only work at the component level. In some cases, users create their own component and for this they would write the equation of that particular component.

The authors' research group has around 20-25 years of model developments using MATLAB®/Simulink® and even more history in simulation codes, in the past using FORTRAN 77 and nowadays rather C++. This constitutes a large amount of available resources that are using a signal-based modeling approach or being somewhat close to that paradigm. For some applications, being able to connect this with more "high-level" type of modeling would be interesting from a practical point-of-view. In the past, various attempts were made to using Modelica/Dymola or the Simulink SimMechanics libraries to model and simulate various multi-body parts of the overall systems. None of these attempts yields to the integration of these tools or modeling approaches into the main modeling workflow of the authors' research group. The additional complexity of mixing the existing tools and models with a separate set of tools and sometimes also quality issues (bugs, instabilities, lack of flexibility, etc.) of the tested software packages yield a non-beneficial additional effort to additional capability/flexibility ratio for the considered applications.

*Scientific Advisor, DLR-FT-FDS, Lilienthalplatz 7, 38108 Braunschweig, Germany, AIAA Senior Member. E-mail: nicolas.fezans@dlr.de
†Research Scientist, DLR-FT-FDS, Lilienthalplatz 7, 38108 Braunschweig, Germany.
§i.e. without the numerous toolboxes which extends its scope and add new modeling paradigms

The present work constitutes a new attempt at extending the signal-based modeling paradigm used, this time with a code-based object-oriented module. Simulink® provides, like many other simulation software packages, the possibly to interface C++ code with a simulation model. This is realized through so-called C++ s-functions. This allows leveraging very complex object structures, algorithms, and all paradigms supported by C++ in Simulink® models. The idea was not to make a very large s-function taking a very large number of inputs from Simulink® and producing a large number of outputs to Simulink®, but rather to design a mechanism permitting to use the capabilities offered by C++ with a series of blocks that would cooperate and share information with each other. The model designers would take the different blocks and place them as needed in their models. By using a set of blocks instead of a large centralized block, the use of the new features feels more natural to Simulink® users: they simply add C++ objects to the model by adding the corresponding s-function blocks. Users already having a working model can adopt the new blockset in a progressive way, so this blockset could serve as a mean to facilitate a transition from a pure Simulink® implementation to either a pure C++ implementation or a mix of both.

Whilst there is a very large body of knowledge in the modeling and simulation community related to multi-body simulation, component-based modeling, Bond graphs, etc., the type of problems and the technical solutions discussed in this paper does not seem to have received much attention. To some extend, the application considered and presented in section III can be compared to the tensor-based writing of flight dynamics advocated by P. Zipfel in Refs. [1–3] and many other of his papers and books. The framework and ideas presented hereafter can be used to build complex object-oriented structures complementing the MATLAB®/Simulink® environment. They can be used to manipulate reference frames and coordinate systems to permit and encourage users to think in terms of tensors even if on Simulink®'s side coordinates still have to be used. The proposed mechanism can also be used for manipulating other objects and not just tensors. The authors fully agree with ideas of P. Zipfel, for instance using tensors in flight dynamics or using C++ over Simulink when performance matters most. They also use self-programmed C++ models (by opposition to code-generated from Simulink) for instance for system identification or for the VIRTTAC (Virtual TesT AirCraft) configuration introduced in Refs. [4, 5]. However, in a larger research group whose members have a wide range of specialties or when experimenting/prototyping, there is added-value in a solution that exploits the best of both worlds: performance, generic programming, and complex objects/behaviors from C++ with the easiness, flexibility, and interactivity of MATLAB®/Simulink®.

In section II, the general-purpose simulation mechanism is explained in a very generic and application-agnostic way. This description might appear very abstract to most aerospace engineers as it requires a certain abstraction level in terms of modeling and simulation concepts as well as familiarity with the C++ language. Section III (page 8) focuses on the concrete features that were developed in the framework of the air-to-air simulation programs developed by the authors. As such, depending on the reader's background and research interests, reading section III before section II or going back and forth between these two sections might be advisable.

doi: 10.2514/6.2022-0808

II. Simulation Mechanism

A. How Simulink interacts with s-functions

The way Simulink® interacts with user-defined s-functions is described in the documentation of MATLAB®/Simulink® in great details[¶]. The application programming interface (API) contains numerous functions and the interested reader is encouraged to read closely the corresponding pages of the Simulink documentation. For conciseness reason, only a limited subset of functions that play or could have played a role for the proposed module is considered in the following:

- **mdlInitializeSizes** is a function that s-functions must implement and where the programmers of the s-function define the inputs and outputs of the s-function (including their number, size of each input/output, datatypes), whether an input as a direct influence on the outputs ("direct feedthrough"), and the number of continuous and discrete states. This function is called very early, before starting the simulation. It permits for Simulink® to know whether the dimensions in the block diagram are consistent with each other and to allocate the necessary memory array(s) for exchanging the data between the block and Simulink® (and indirectly with the other blocks).
- **mdlStart** is a function that allows initializing the block at the start of the model. Ressources (e.g. memory) are typically created, allocated, opened, or bound in that function.

[¶]This information has been provided in the documentation at least since version R2007b. The exact location in documentation has changed over time. It can currently be found on the page called *Simulink Engine Interaction with C S-Functions*, which can currently be accessed online at the address https://www.mathworks.com/help/simulink/sfg/how-the-simulink-engine-interacts-with-c-s-functions.html.

- **mdlTerminate** is the counterpart of `mdlStart`. Ressources created, allocated, opened, or bound in mdlStart will typically be released in mdlTerminate.
- **mdlOutputs** is the function that Simulink® calls at every step of the Simulation to obtain the s-function block outputs for the current inputs and states.
- **mdlDerivatives** is the function that Simulink® calls at every step of the simulation to obtain the derivatives of the s-function states for the current inputs and states, when the s-function has continuous states. None of the s-functions used in the present work has states (neither continuous nor discrete), thus mdlDerivatives plays no role hereafter. Note that it is not a limitation of the approach shown in this paper, but rather a choice that was made to keep the design as simple as possible, at least for now.
- **mdlUpdate** is similar to `mdlDerivatives` but for discrete states. It is also not used in the present work.

B. Four different roles for the additional blocks

The blockset is composed of s-function blocks which:

- consist of a single C++ source file, which implements the required functions for the Simulink s-function API and manage the interaction with Simulink® (e.g. configuration parameters), and
- are linked with a static library (the same one for all blocks) which provide the core functionalities.

A mask is included on most blocks to ease their configuration by providing explanations to the user and increasing the affordance of these blocks by letting users directly select possible configurations with popup menus and checkboxes. Mask callbacks are also used to make the appearance of the blocks indicate their current configuration, such that it is never required to open the mask GUI to know how the block is configured. Many standard Simulink® blocks change their appearance based on their configuration, such that this feels very natural to Simulink® users.

At the core of the proposed Simulink® extension is a C++ class called `World` which manages all elements that can be added to the simulations. Various types of elements can be added to the `World` object: a common interface is provided using a virtual class called `WorldElement` that each type of world elements inherits from. At the start of the simulation, the very first step that is needed is to create the `World` object that all blocks from the blockset will use. For this a specific block is required: its underlying s-function creates the `World` object, makes it permanent (i.e. until the end of the simulation), and stores its address in its Simulink® `PWork` vector. This s-function needs to be executed before all other s-functions from the blockset. This can be ensured by establishing signal connections between the s-function blocks: the s-function that creates the `World` object takes no input signal and has one output that is connected (directly or indirectly) to all other s-functions from the blockset. With this interconnection topology Simulink® is forced to call that s-function first. Figure 1 shows a block-diagram representing the interconnections between the different types of s-functions and with the rest of the Simulink® model.

The signal that is connected to the following s-functions is also used to transmit the pointer address of the instantiated `World` object to them. In reality, in order to make the overall system robust to errors in the interconnection of the blocks, all the pointers to the underlying C++ objects passed from block to block via signals (in orange in Fig. 1) are actually pointers to wrapper objects that allow checking the type of the objects passed. If users wrongly connect the blocks, the simulation would stop immediately and provide useful error messages instead of crashing due to some segmentation fault. In addition to the `World` object, a class called `UnsynchronizedWorld` also exists. It has two roles: it ensures that the `World` data and features cannot be used in a possibly undefined state (e.g. before all required updates at the current time were taken into account) and it provides an API to declare new objects and send update requests at each time step (e.g. new coordinates for an object, strength of a wind field, etc.). In terms of object-oriented design patterns, this class is technically rather a façade to the `World` class, but it might be helpful for users to rather think of it as a proxy class which controls the access to the `World` object and prevents the "client" s-functions to work with it as long as all elements have not been synchronized for the current time step.

One particular s-function is responsible for triggering the synchronization mechanism. At every time step, upon completion of the synchronization, it provides a direct access to the `World` object address via its output. All blocks connected to that output are necessarily called by Simulink® after the synchronization took place. They can use the methods of the `World` class to request a wide range of computations or to extract information from the `World` object and all its children.

doi: 10.2514/6.2022-0808

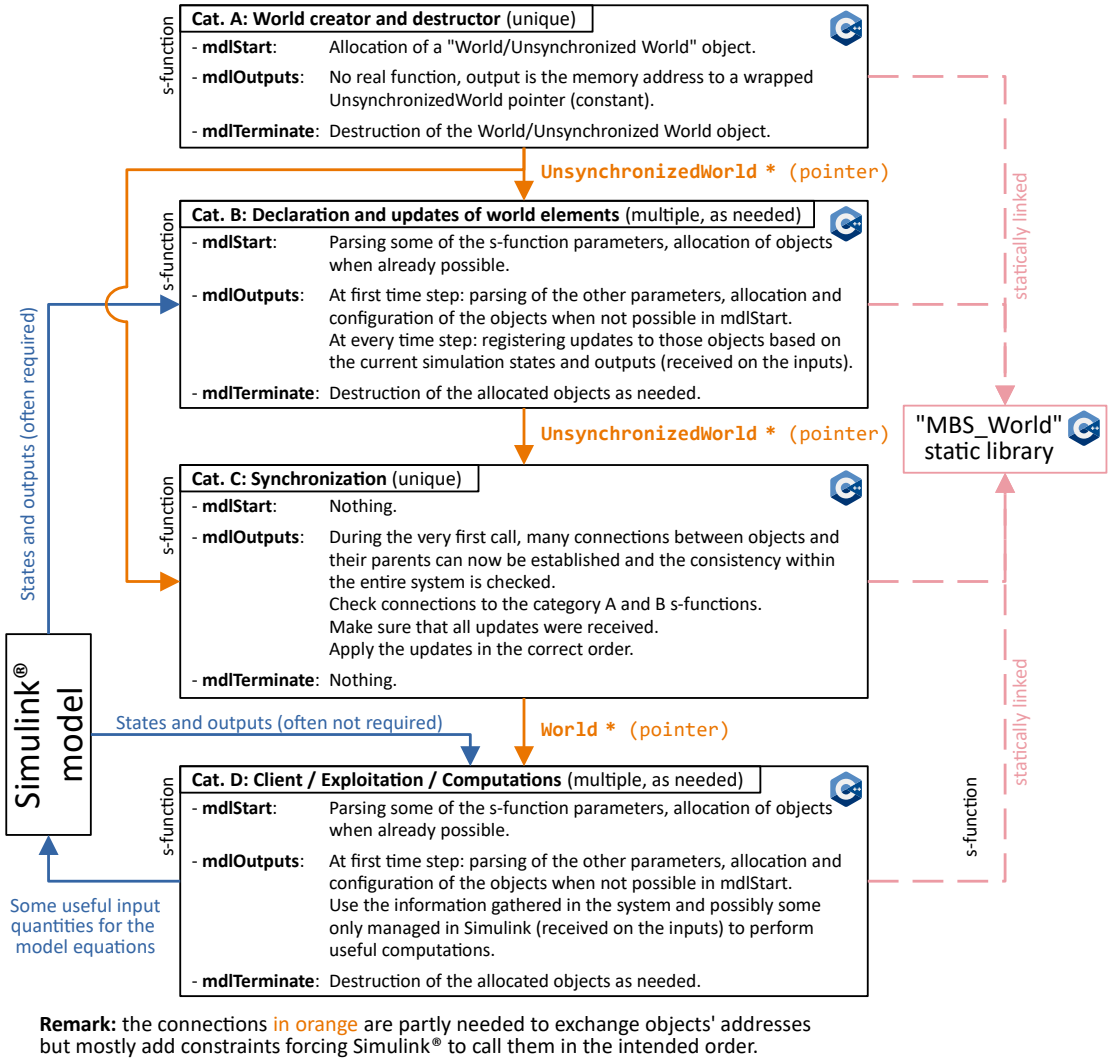


Fig. 1 Interconnections between the different types of s-functions and the rest of the Simulink® models. For simplicity, only one category B and one category D s-function block is represented: further category B or D s-function blocks would be connected in parallel to the one represented for this category.

At the end, the various s-functions mentioned can be decomposed into four categories, based on their roles and responsibilities:

- **Category A: Creation of the world** – Only one extremely simple s-function ⇒ almost never needs to be updated.
- **Category B: Declaration of world elements and their management** – Several different s-functions, each possibly used multiple times in the model. They create concrete objects inheriting from `WorldElement` and send updates to these objects at each time step. As long as users do not want to create fully new types of world elements or to create new options for the existing ones, they will not have to interact with the s-function source code. In that case they only interact with the Simulink® block, connect it to the other blocks, feed them some signals in input, and configure them through their mask.
- **Category C: Synchronization of all world elements** – Only one simple s-function which calls the synchronize method from the `UnsynchronizedWorld` object ⇒ almost never needs to be updated. Its input interface needs to match the number of declaration blocks + 1 (connection from the category A s-function).
- **Category D: Client / Exploitation** – Various s-functions performing different tasks by leveraging the information contained in the `World` object and its children or by leveraging the features (e.g. algorithms) implemented in the library.

doi: 10.2514/6.2022-0808

C. Why decomposing the C++ code in several s-functions? Would it not be possible to integrate everything in one large s-function?

The numerous s-functions described above are called sequentially by Simulink® during a simulation step. The first s-function (at the top of the diagram of Fig. 1) creates the `World` object and only plays a role at the very start of the simulation. All the other ones from category B are called, then the synchronizer (category C), and finally the client s-functions (category D). Note that in reality only three scheduling constraints in the order of the calls are required:

- The category A s-function must be called before all others.
- Every s-function from category B must be called before calling the synchronizer (category C).
- Every s-function from category D is called after the synchronizer (category C).

All s-functions are having direct feedthrough, yet no algebraic loop is created. The category A s-function has no input (i.e. not part of a loop). Category B s-functions work on pure inputs (e.g. constants) or on values that are either states or derived from them. The category C s-function is directly connected to the outputs of all category B s-functions. The category D s-functions are also all taking the output of the category C s-function as input. So there is a chain of direct feedthrough s-functions involving the paths $(B_i)_{i\in[\![1,m]\!]} \rightarrow C \rightarrow (D_j)_{j\in[\![1,n]\!]}$. However, the outputs of all s-functions from category D are only for computing outputs of the model (i.e. not being involved in paths looping back the s-functions from categories B or C or themselves) or for computing derivatives of states that are used in the category B s-functions. Algebraic loops cannot be created by using these s-functions as intended. In the concrete example shown later in section III, the category B s-functions take states like position, orientation, velocities, and angular rates as inputs. In this example, the outputs of the category D s-functions are used in the computation of forces and moments, which are used for computing the derivatives of the velocity and angular rate states. All paths from a category D s-function to a category B s-function therefore pass through a differential equation of the model.

Based on this analysis of the interconnections and of the possible paths connecting these s-functions, it is clear that the entire C++ code could have been called from one large s-function. It would do the same things than the category A s-function at the start and at then end of the simulation. During the simulation loop, it would perform all operations from the category B s-functions first, then perform the synchronization, and finally perform all operations from the category D s-functions and provide the computed quantities to Simulink® for further use in the model equations. In some cases, and especially at the very first time step, the underlying code could be simplified by merging everything into one large s-function. However, that would massively impact the signal routing in the model. This would also prevent from having some of these s-functions at the most suited place in the model or being even part of a library that is reused within the model.

doi: 10.2514/6.2022-0808

Integrating everything into one large s-function would also complicate the configuration of that s-function and its interfaces would constantly have to be adapted to the current configuration. In contrast, with the decomposition in several s-functions, users can add a new world element to the model by simply adding one of the category B s-function blocks without having to change the s-functions or having to recompile them[‖]. If system modelers develop various variants of a model, integrating everything into one s-function would also make the system significantly harder to maintain. With separate s-function and blocks, each model can include different combinations of blocks with individual configurations without having to manage variants in the C++ code base. A typical use-case for that in the application example shown in section III is the case where a new wind field is added to the environment, e.g. a wake vortex or a gust. The corresponding block is added and configured and every client s-function (category D) of the model that requires wind data receives the modified wind automatically. Regardless of the computations to be performed (e.g. aerodynamics, air data sensors, lidar sensor, etc.), no change in the source code is required to account for the modified wind.

Both solutions (separate s-functions and unique large s-function) have advantages and disadvantages: the authors opted for the decomposition in separate s-functions, because this feels more natural to Simulink® users and because this create well-defined modules with clear boundaries and that are easier to test and validate. Model developers, not always at ease with the C++ language, can more easily start developing new blocks without having to integrate their code in the underlying "core" library[**]. By using masked s-functions, the complexity of the code of the library is mostly hidden behind a high-level API allowing to use most features by writing only a small amount of fairly simple code in one of the s-functions or even by simply configuring one of the existing blocks.

[‖] Only for desktop simulation. For compiled real-time models, a new source code must be generated for the modified Simulink®, this code must be compiled, and the executable deployed.

[**] The library currently consists of 38 classes and about 12,000 lines of code with rather non-trivial interactions between its parts. That code might significantly grow in size with the introduction of new types of world elements and by adding new algorithms to support the need of the client s-functions.

D. Why passing raw pointers via Simulink® signals between the s-functions?

One of the most disputable design decisions in the herein described blockset is the fact that raw pointer values are passed between s-functions (orange connections in Fig. 1). Two main objects are made accessible to the other functions this way: a raw pointer to the UnsynchronizedWorld object (between category A and B s-functions as well as between category B and C s-functions) and a raw pointer to the World object (between the category C s-function and the category D s-functions).

Sending raw pointers through Simulink signals gives the possibility for users to crash MATLAB®/Simulink® entirely, by providing fully invalid addresses or objects. Whilst giving the possibility to users to cause a crash, at least this type of user mistake leads the system to fail fast: the mistakes do not remain undetected and corrective actions can be taken quickly. More subtle errors would be largely more problematic and source of concern. Besides, end-users hardly ever touch the connections that could lead to such crashes. Once new blocks have been connected successfully, there is no reasons for end-users to modify these connections.

Hiding the interaction, e.g. through shared memory areas, would make it harder for users to crash the simulation. It would also make it slightly harder to check that all category B s-functions were indeed called before calling the synchronization block (category C) and that all category D blocks are called after the synchronization. The first version of the library and blockset ran on old 32-bit MATLAB® versions and the pointers were cast as unsigned 32-bit integer and routed as such through the model. The used simulation models almost exclusively use doubles and very few floats or integer types in the global input and output interfaces, so inadvertently creating an erroneous connection that would not be caught by the data type mismatch was very unlikely. Later, the system was ported to 64-bit versions: the 64-bit pointer addresses are casted to doubles as Simulink® did not support 64-bit integer types prior to version R2019a. Using doubles as signal types to transmit raw pointers drastically increased the risk of having users establishing wrong connections and crashing MATLAB®/Simulink® due to memory segmentation violations. At the time of the switch between 32 and 64-bit, the main use case was already established and the few users of these blocks understood the inner-working of the blockset to avoid such mistakes.

doi: 10.2514/6.2022-0808

Passing pointers of the wrong type is another type of mistake that users could make. This case is checked and reliably caught in the s-functions. In that case a crash of MATLAB®/Simulink® is prevented, the simulation stops right at the start and an error message explaining the error made and how to fix it is provided in the Simulink® error dialog window[††]. This check is implemented using dedicated wrapper object classes (the only real C++ object type whose addresses are sent or received by the s-functions) and by leveraging the C++ run-time type information (RTTI). The pointer to the real object is contained in a member variable of the wrapper object and the receiving s-function can use the RTTI to perform the type checking. Readers attempting to reproduce the described mechanism on Windows can find a couple of useful hints on the use of RTTI in mex/s-functions under Microsoft Windows in Appendix A.

The described Simulink® extension is an in-house development with only internal users who are all working in the same department and building as the authors, i.e. with easy access to support if needed. Until now, no user has ever triggered a crash of Simulink® by making wrong connections between the blocks. Upfront explanations and the explicit labeling of the input and output ports have proven to be sufficient in the context of in-house users. The interconnections between the blocks are also set up once in the model and almost never touched again afterwards, such that users are not really risking to trigger such errors while simply making other modifications in the model. Nevertheless, that particular design aspect has been identified as potentially problematic and is likely to be improved with one of the next versions. Currently, the code does not use specific features or implementations from the operating system and does not require non-standard libraries: switching to other solutions, e.g. using shared memory, would certainly introduce such dependencies. Using the Boost.Interprocess library could be considered for creating an alternative to the current solution of passing raw pointers via Simulink signals. On a side note, the current signal-based transmission of pointers between the s-functions would also allow to instantiate several *worlds* working as parallel universes. Whether there are real practical use cases for simulation models with such parallel universes remains unclear.

[††] or printed in the console, if the simulation was started programmatically

E. Setting up the World at the start of the simulation

As described above in section II.A, Simulink® interacts with s-functions via a series of functions that are part of the API and are either mandatory or optional. Overall, the s-functions involved in the blockset behave fairly normally, but a couple of elements regarding to the start and initialization of the simulation are worth mentioning.

Delayed initialization of world elements

The `World` object is created when the `unsynchronizedWorldCreator` s-function's (which is guaranteed to be the first to be executed) `mdlStart` method is called. After that, the various other s-functions' `mdlStart` methods are successively called as well. During these calls, other objects might be allocated, initialized (e.g. reading data from files etc.) and their address retained via the `PWork` vectors. Initialization status might be tracked with the `IWork` vectors as well. To this aim, often s-function parameters are used: they are also validated and partly applied during the `mdlStart` calls.

The penultimate step in the configuration of most `WorldElements` often requires initial state values, which are first available at the first time step, when the `mdlOutputs` method is called. At that stage, the registration of the `WorldElement` can be initiated by calling the corresponding method on the `UnsynchronizedWorld` object. At that point, these registrations cannot be validated and completed yet. Most elements declare relationships with other elements by referencing them via their names. Those elements might not have been registered or even been allocated yet as no order is prescribed in the calling of the category B s-functions. Only when the `mdlOutputs` function of the synchronization s-function (category C) is called for the first time, it is certain that all `WorldElement` registrations should have been received. All these registrations are analyzed following a multi-pass procedure. As soon as all prerequisties for a `WorldElement` registration are satisfied, this registration is processed and the element is added to the World. This process can only fail if a `WorldElement` refers to another `WorldElement` but that `WorldElement` was not declared (e.g. the user forgot it or deleted the s-function block by accident). In that case, the simulation is ill-formed and the process stops with an adequate error message explaining which block had declared an invalid relationship, naming the `WorldElement` it referred to and whose declaration is missing. The world elements can only declare relationships to *parents*, which implies that the relationships can only define trees (no cycles can be created). As it will be seen later in the concrete example, several different trees can co-exist in parallel, because various types of the relationships can be defined and for a given element the *parent* element might be different for each type of relationship.

doi: 10.2514/6.2022-0808

Will `mdlTerminate` be called?

The standard simulation loop with Simulink® foresees that the entire simulation model is set up, initialized (among other things by calling `mdlStart`), then calls are made by for integrating the model equations (`mdlOutputs` and `mdlDerivatives`/`mdlUpdate` when the s-function contains continuous/dicrete states), and finally the simulation ends and the functions `mdlTerminate` and `mdlCleanupRuntimeResources` can release the used resources (e.g. allocated objects, opened files, network ports, etc.).

The described extension is used both on the desktop as well as after code generation and compilation in the DLR AVES simulator [6, 7]. When using large simulation models in a flight simulator, large amounts of data might be allocated at the start of the simulation. Simulation experiments often consists in several simulations: pilots are presented with a given scenario, at the end the simulator is reset to the same or other initial conditions, and another simulation scenario starts. If the same resources are needed between successive runs, the simulation might be configured such that no call to `mdlTerminate` is made when resetting the simulation. This is the case with the DLR AVES simulation environment. In that case, the s-function programmers need to keep track of the resources used and determine whether the simulation that is starting should re-use existing resources or whether new resources need to be created, bound, or reserved. In the case of memory allocation, this is required in order to avoid memory leaks by repeatedly allocating memory for these resources at each new start of the simulation and never deallocating the previously allocated ones. For other types of resources, a second allocation with the same parameters might simply fail[‡‡]. Note that allocating new objects is not only an issue due to always increasing memory needs: this can also lead to undefined behaviors if some pointers still point to the old objects while others point to the new ones. In the worst case, the entire simulation program might crash as some variables are used to remember whether the created objects were properly initialized or not (see above the delayed initialization): depending on how the tests are implemented, new objects might be created but not fully set-up prior to their first use, because the s-functions might wrongly believe that they were already set-up and not proceed to the final initialization steps for the newly created objects.

[‡‡]e.g. both objects trying to access a peripheral that does not allow concurrent accesses

III. Application to Air-to-Air Refueling Simulation Models

DLR is working on the modeling, simulation, and automation of air-to-air refueling maneuvers with the probe-and-drogue system [8, 9]. In Ref. [8] (section III.B.), the authors already mentioned that a Simulink® model extension for relative kinematics, wind fields, and aerodynamic interaction had been developed to bring multi-body-simulation-like features to the developed Simulink®-based aerial refueling simulation programs. This section provides a significantly more detailed description of the developed features and how they are used in these simulation programs. These features were implemented using the approach presented above by defining a series of *world elements* for the different main objects, by implementing various algorithms to be able to provide a high-level declarative syntax to the end-users and taking care of the details behind the scene, and by implementing a series of helper classes for simplifying the implementation of these algorithms.

A. Basic Tools

In order to simplify the implementation of all other features, a series of template classes has been programmed to provide a significant number of basic mathematical functionalities including vectors, matrices, linear algebra (QR/LUP/Choleski, solving linear systems, MATLAB®-like backslash operator, etc.), multidimensional template-based lookup-tables, and homogeneous transformations. These classes are not necessarily very complicated but are heavily used in the code of the other modules.

Very good C++ libraries exist and could certainly be used to replace some of the in-house developed classes, especially for the linear algebra features. However, the in-house classes are largely good enough and are designed to integrate nicely with all the other classes of the module and feel familiar to MATLAB® programmers. The high-quality open-source libraries available usually provide a much larger set of features than what was required in that project. The needed features were also easy to implement and implementing them prevented the authors from having to manage code from different sources and with different licenses and programming styles. Integrating external libraries remains an option for future evolutions.

doi: 10.2514/6.2022-0808

B. Kinematics and Dynamics

In addition to the basic tools for performing mathematical operations, reading/writing files, etc. a significant portion of the currently available classes deals with relative kinematics problems and the implementation of wind fields.

1. Reference Frames, Points, Coordinates

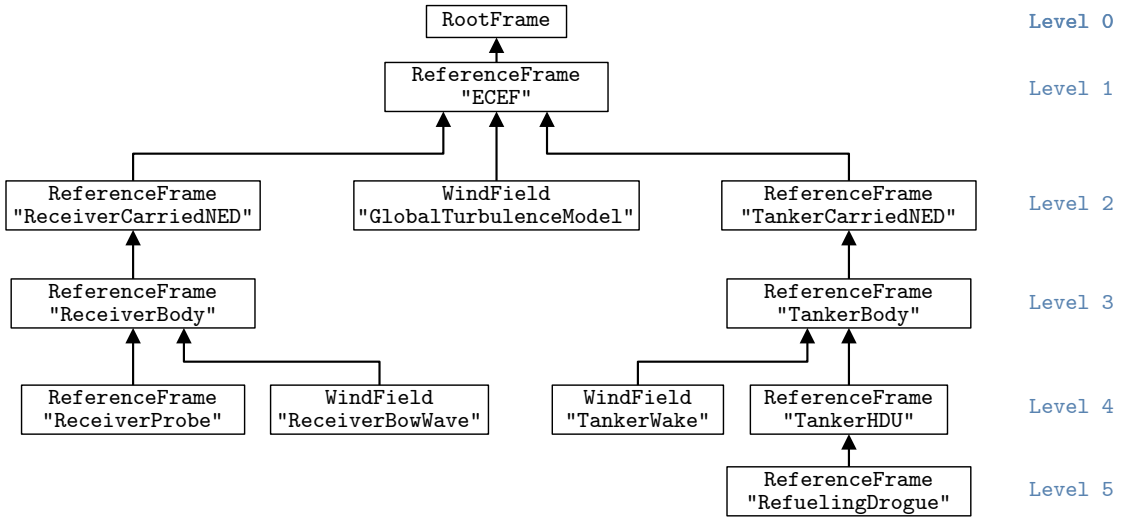
A class called `ReferenceFrame` allows defining individual reference frames and is one of the most heavily used object type throughout the entire code. Most of the world elements registered in the "World" object in the aerial refueling simulation programs are reference frames. Each reference frame possesses: a unique name, two parents (one for its pose[§§], and one for its twist, cf. section III.B.2), a relative pose with respects to its pose-parent, a relative twist with respect to its twist-parent. Considered as a whole, the relative pose and the relative twist correspond to the six degrees of freedom describing the kinematic relationship between two bodies in a three-dimensional space. Upon receiving new state values via the inputs, the new relative pose and twist are saved in separate member variables until the synchronization of all received updates can be made. When the synchronization is triggered, two virtual methods from WorldElement called `performUpdate()` and `performTwistUpdate()` are called on each world element. The appropriate order for these calls is to visit all nodes of the pose respectively twist relationships tree in a bread-first order: at each level the parents must be updated before their children. Fig. 2 shows a typical tree structure as used in the air-to-air refueling simulation programs. To keep this graphical representation readable, only a subset of the world elements[¶¶] are included. Comparing the structure of the pose (Fig. 2a) and twist (Fig. 2b) trees, it appears that reference frames which are not fixed with respect to their parent often have ECEF (Earth-Centered-Earth-Fixed) as *twist parent*. The reason for that is that velocities and angular velocities are often expressed with respect to an assumed inertial / Galilean system. In this simulation program, ECEF is considered as fixed against the *root frame*[***] and as common ancestor of every object of the *world*. This choice is adequate for the considered application to air-to-air refueling, but

[§§] ReferenceFrame inherits from WorldElement, which already has a parent: the parent property from WorldElement is the pose parent for ReferenceFrame.

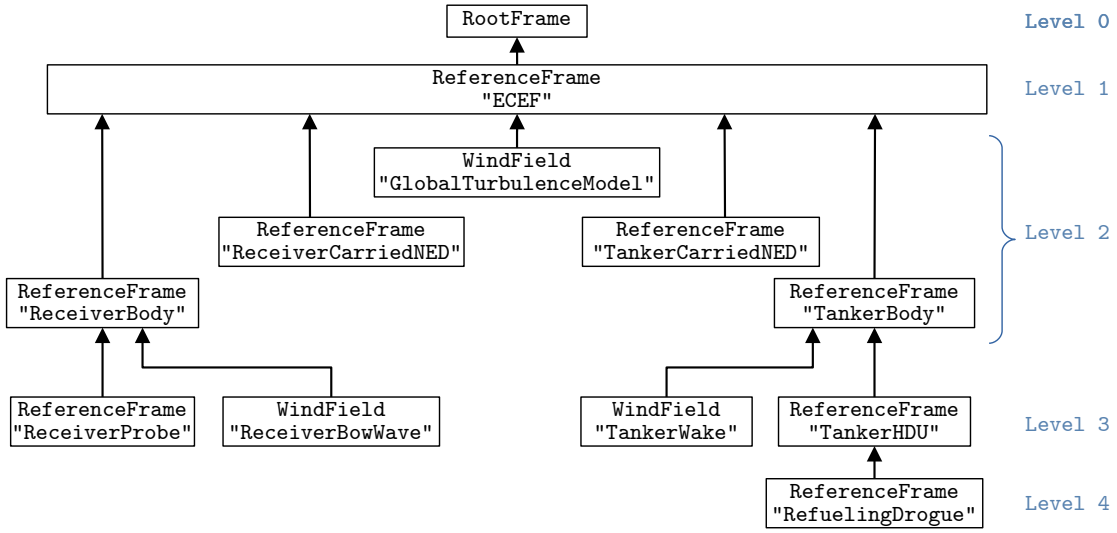
[¶¶] Remark: some nodes here are wind fields, cf. explanations in section III.C, and not reference frames.

[***] The *root frame* is named `RootFrame`: this name is reserved and cannot be used for other world elements.

the module used in this work is generic and any reference frame could be attached to the *root frame*[†††].



(a) Pose relationships



(b) Twist relationships

Fig. 2 Pose and twist trees for a few of the main reference frames and wind field elements used in the considered air-to-air refueling simulation. Both trees consists of the same *world elements* but have different structures, because, in this example, the user decided to describe the twists (translational and rotational motion) of the body coordinate systems (`ReceiverBody` and `TankerBody`) directly with respect to `ECEF` instead of expressing them with respect to `ReceiverCarriedNED` and `TankerCarriedNED` frames, respectively.

[†††]For instance using J2000 for a satellite, ICRF (International Celestial Reference Frame) for an interplanetary mission, or a local Cartesian system with flat and fixed Earth assumption for a small UAV.

doi: 10.2514/6.2022-0808

Coordinate systems are implicitly defined with each reference frame. For position coordinates, the default set of coordinates are right-handed Cartesian coordinates, but other options such as cylindrical, spherical or WGS-84 are available. The list of different options is defined via a C++ enum. The enum value is not attached to the reference frame, but is carried along with any object that contains such coordinates[‡‡‡]. The fundamental difference between a reference frame and a coordinate system can be explained on the example of the Earth-Centered-Earth-Fixed (ECEF) reference frame. ECEF is defined only once, but a relative position with respect to ECEF might be expressed with Cartesian coordinates in some places of the code and with WGS-84 coordinates in other places: one reference frame but potentially several sets of coordinates.

Users only need to specify the type of coordinates when injecting new coordinates in an object (in that case to declare how these numerical values must be interpreted) or when requesting coordinates. All mathematical operations occurring behind the scene, check that the coordinates are compatible and convert them if necessary. The default conversion prior to mathematical operations usually consists in converting the non-Cartesian coordinates to Cartesian coordinates (usually most efficient choice for the following computations). Three types of relative orientations are currently supported: Euler-Cardan angles, quaternions, and a special case for WGS-84 North-East-Down system. The latter case was introduced to prevent inconsistent values when working with WGS-84 coordinates, which is very common in aeronautics. The orientation of a North-East-Down (NED) system with respect to ECEF only depends on the position of its origin and can therefore be derived from it. This option was therefore introduced to simplify the definition of NED systems, otherwise users would have had to determine Euler-Cardan angles or quaternion values matching the current set of WGS-84 coordinates and could have gotten it wrong. In a way, this special case is similar to typical camera constraints in 3D graphics (e.g. camera rotates to always point toward a particular object).

Behind the scenes, the transformations between coordinates, compositions of motion, etc. are implemented with specialized three-dimensional and four-dimensional vectors and matrices. In most cases, homogeneous transformations are used to tackle both rotations and translations in one step. To automatically determine the transformations required, the relationship between any combination of two reference frames that are not respectively parent and child can be determined, if needed. The fact that all coordinate systems are part of the same tree is exploited for this. When building the tree, the level at which a node was inserted is also saved in the node itself, because it is useful to know later for some of the search operations.

doi: 10.2514/6.2022-0808

The relationship between two world elements can be identified with a simple search strategy:

1. The search starts with the two elements A and B, whose relationship needs to be determined. If the two elements are the same (pointers to the same world element), then the search is over: they are equal.
2. If the two world elements are not the same, then the respective levels l_A and l_B are compared:
   (a) In case one of the elements has been inserted in the tree at a higher level than the other (say $l_A > l_B$), then there is a chance, that the other element (B) is one of A's ancestors. Go up in the branch containing A until reaching level l_B and go to step 1.
   (b) If both elements are at the same level but not equal, than we go up one level in both branches and go to step 1 again to check whether they share the same parent.

During this search process, all world elements encountered in both branches of the tree are saved into two lists (one growing forwards and the other backwards). When a common ancestor is found, both lists are stitched together to form the full path from one element to the other one. This process cannot fail as all world elements are connected in a tree structure: in the worst case, the common ancestor will be the root of the tree.

For performance reasons, a manually-managed cache is used, allowing the memoization of the functions searching the relationship between two elements and computing the combined transformation. The search and the computation of the corresponding coordinate transformation matrices is skipped if they were already computed. If two coordinate systems are not having a relative motion (property that can be set for each children, with respect to its parent), the matrices are computed only once, at the first time step. For coordinate systems that do move with respect to each other, the computation is performed the first time that this path and homogeneous transformation matrices are needed during a given (major or minor) time step and the result is cached. If the same transformation is needed again during the same time step, then the cached result is directly reused.

[‡‡‡]This forces to manipulate slightly larger objects (the enum value being added), hence causes a slight overhead, but potentially prevents from misinterpreting and, as a consequence, from misusing the numerical coordinates in the recipient's code.

2. Relative Pose, Relative Motion Twists, and Efforts Exchanged Between Bodies

As mentioned in the explanations of the reference frames, the current states of the six degrees of freedom motion of each reference frame (position, orientation, velocity, angular velocity) are being registered, updated/manipulated, and are used for performing many computations without having the final users having to think about the required coordinate transformations (both between reference frames and between types of coordinates e.g. Cartesian, WGS-84, etc. for positions or Euler angles, quaternions, etc. for orientation). The formalism used inside the module is very common in robotics, but significantly less known and used aeronautics. The way relative positions and orientations are expressed is fairly standard, but they are manipulated jointly as *pose* objects. When working with numerical values, the corresponding coordinate system needs to be known, which is why each of these objects carry three additional member variables: 1) a pointer to the reference frame object, 2) a C++ enum value indicating the type of position coordinates (see above), and 3) a C++ enum value indicating the type of orientation coordinates. This ensures that anywhere in the code, these objects can always be interpreted properly.

A similar concept is used for the relative motion (velocity and angular velocity) and the efforts (forces and moments). Here the formalism known under the name *screw theory* is used [10]. The relative motion of one (rigid) body S_2 with respect to another body S_1 can be described with three elementary pieces of information:

1. an angular velocity vector $\overrightarrow{\Omega}$,
2. a (translational) velocity vector $\overrightarrow{V_A}$ for a given point A
3. the position/definition of point A (called *velocity reference point* or VRP) expressed via the position vector $\overrightarrow{OA}$ with O being the origin of the coordinate system used.

When working with numerical code, here again a clear definition of the coordinate system, in which the coordinates of these vectors and points are expressed, must be carried with that object. Currently, for velocities and angular velocities only Cartesian coordinates are considered and all vectors need to be expressed in the same coordinate system. Therefore, only a pointer to the corresponding reference frame needs to be added to make that object self-explaining. In the formalism of the screw theory, this constitutes a ***twist***. The two most common transformations that are applied to twists are:

- **Expressing the twist in a different coordinate system:** This is done by applying the required rotations of the angular and translation velocity vectors, and by transforming the position vector into the new coordinate system. The latter is done via a homogeneous transformation, as the origins of both coordinate systems can be different. Note that the expression in a different coordinate system is not a composition of movement: the twist still describes the motion of S_2 with respect to S_1, only using different sets of coordinates.
- **Changing the definition of the velocity reference point:** For expressing the velocity vector in the new VRP B the following formula is used: $\overrightarrow{V_B} = \overrightarrow{V_A} + \overrightarrow{BA} \times \overrightarrow{\Omega}$. The twist expressed in B therefore contains the same angular velocity vector Ω, the velocity vector $\overrightarrow{V_B}$ at point B, and the position vector $\overrightarrow{OB}$.

These elementary transformations on twists form the basis for all other transformations, e.g. composition of motion. For instance, the twist describing motion of S_3 with respect to S_1 being easy to express based on the twists describing the relative motions of S_3 with respect to S_2 and of S_2 with respect to S_1, provided that the coordinate systems used are the same and that the same VRP are used as well. The advantage of using such formalism becomes obvious when manipulating more than a couple of systems: any relative motion being decomposed as a series a simple transformations that can easily be combined in a loop and whenever these transformations are defined with different coordinate systems, the corresponding transformations are automatically determined and performed.

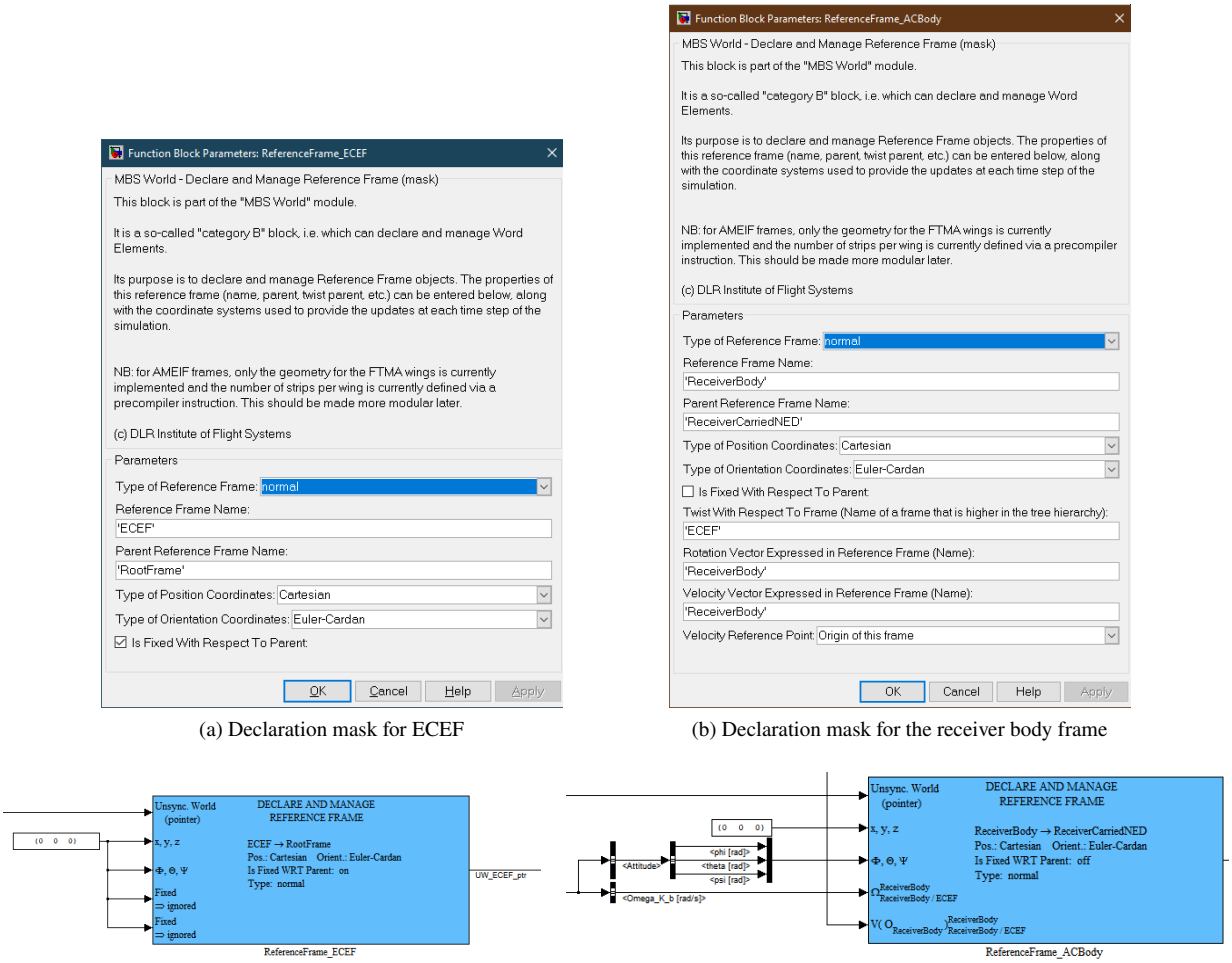
Another type of screws, called ***wrench***, works very similarly but expresses efforts between two systems, instead of relative motions. Just as twists, wrenches need three elementary pieces of information:

1. a force vector $\overrightarrow{R}$,
2. a moment vector $\overrightarrow{M_A}$ for a given point A
3. the position/definition of point A (called *moment reference point* or MRP) expressed via the position vector $\overrightarrow{OA}$ with O being the origin of the coordinate system used.

The definition of a wrench should feel more familiar to aeronautics engineers as any aerodynamic data set already refers to the concept of moment reference point. The wrench object is simply a formalism that holds and carries these quantities as one object, which helps preventing errors by combining things that could not be combined (e.g. not the same MRP or not the same reference frame for the coordinates). The same transformations (as for any type of screws) apply for wrenches as well: $\overrightarrow{M_B} = \overrightarrow{M_A} + \overrightarrow{BA} \times \overrightarrow{R}$.

doi: 10.2514/6.2022-0808

doi: 10.2514/6.2022-0808



(a) Declaration mask for ECEF

(b) Declaration mask for the receiver body frame

(c) Block appearance for ECEF

(d) Block appearance and connections for receiver body reference frame

Fig. 3 Example of configurations and appearance of the developed blocks

3. Simulink® Block Supporting the Definition of Reference Frames

The appearance of the block permitting to declare and manage a reference frame changes to reflect the current setting of the block. Fig. 3 shows some of the blocks as configured in the air-to-air refueling programs. The first input is used for the pointer to the UnsynchronizedWorld object. The other four inputs allow passing vectors for the position, orientation, velocity, and angular velocity. Depending on the configuration of the block, the labels give a visual cue to the user about the expected inputs. In some cases, no information is required: in that case the port is not removed, but the label explicitly states that this port is ignored. Removing the port would be possible, but is avoided as this changes the structure of the system and causes issues when used inside a Simulink® library. This happens when a coordinate system is declared as fixed (with respect to its parent): in that case velocity and angular velocity vectors are not needed as they can be deduced from the motion of its parent and their respective locations (cf. Fig 3c for ECEF or the ReceiverHDU block in Fig. 3d). This also happens for the orientation in the case of the WGS-84 North-East-Down orientation, cf. ReceiverCarriedNED (top block in Fig. 3d).

The velocity and angular velocity vector inputs of the blocks shown in Fig. 3 explicitly show the current configuration. For instance:

$$\mathrm{V}(\mathrm{O}_{\mathrm{ReceiverBody}})^{\mathrm{ReceiverBody}}_{\mathrm{ReceiverBody\ /\ ECEF}}$$

expresses that the expected input is the velocity of point $\mathrm{O}_{\mathrm{ReceiverBody}}$ (i.e. the velocity reference point is the origin of the reference frame ReceiverBody), as point belonging to ReceiverBody, with respect to ECEF and as a set of three

components expressed in the ReceiverBody Cartesian coordinate system (the expression frame being the exponent in this notation). For the expression frames provided as parameter of the s-function block, the present block mask only considers the Cartesian system associated to a reference frame. This was the only option that was needed until now, but a new parameter could be provided in the GUI of the block mask to let users use the other options (e.g. provide a three-dimensional velocity as time derivative of latitude, longitude, and geodetic height). A similar representation is used for the angular velocity vector, but, as explained in section III.B.2, it does not depend on a reference point:

$$\Omega^{\text{ReceiverBody}}_{\text{ReceiverBody / ECEF}} \ .$$

In the middle of the Simulink® block (cf. Fig 3), the element name and post parent frame name are indicated respectively left and right of the arrow (→) symbol. The type of position and orientation coordinates is indicated as well, which is redundant with the corresponding labels for the input ports but improves clarity. In the case of Cartesian and WGS-84 position coordinates (respectively noted 'x, y, z' and 'lat, lon, h') most users will probably know what they mean. As the symbols used for spherical ('r, ϕ, θ') and cylindrical ('ρ, ϕ, z') coordinates might be less easy to interpret, it was decided to write this as well in the middle. Finally, whether a coordinate is declared as fixed with respect to its parent is also reminded explicitly in the middle of the block.

This particular block is shown in more detail than the others because this allows displaying the type of high-level declaration (e.g. simply referring to other reference frames by name) that the proposed Simulink® extension makes possible. Better graphical representations can certainly be developed to improve the human-machine interaction and reduce the risk of errors. As developer of such blocksets, managing the functionality in the source code of the s-function and the configuration and appearance of the GUI separately with MATLAB®/Simulink® feels quite inefficient and error-prone. Besides, manipulating the appearance of the block based on GUI parameters and callbacks often poses issues when using these blocks within libraries. Having all possibilities of a block mask accessible with optional methods of the C/C++ s-function would be a very attractive solution. Simulink® could ask via callback functions for the name of the block type, the help text, the name and type of the parameters, the display commands, etc. By giving the option of managing everything in one place (the s-function code), the version management would be greatly simplified.

The screenshots shown in Fig. 3 also show a "type: normal" indication in the middle of the block. This indication is related to the first parameter in the mask GUI. The type *normal* corresponds to the standard `ReferenceFrame` object, as described in this section. Some derived classes of `ReferenceFrame` can be instantiated via this block as well, which allows sharing the same s-function and avoiding code duplication. One such derived type is called `AMEIFFrame` and explained later in section III.D.

doi: 10.2514/6.2022-0808

C. Wind Fields

One of the desired capabilities that was defined before starting to develop the described Simulink® extension was to have a flexible and powerful way to define wind fields. During air-to-air refueling maneuvers, the receiver aircraft is almost always flying in close proximity to the tanker and being influenced by its wake [9, 11, 12]. The wake of the tanker therefore needs to be modeled properly and integrated in the simulation programs. It influences not only the receiver aircraft, but also the refueling systems (e.g. hose and drogue, or boom in the boom-and-receptacle system). The presence of the receiver aircraft also influences the refueling systems, especially the so-called bow wave [13–16] plays a major role during the coupling of the probe-and-drogue system. The probe has a limited length such that the drogue enters the bow wave during the final phase of the contact maneuver. The bow wave pushes away the drogue such that, without active control actions in the very last phase of the contact maneuver, the drogue would systematically move away before the probe could make contact with it.

The tanker wake constitutes a wind field that is most suited to define in the tanker body system or at least in a tanker-carried coordinate system. In contrary, the receiver bow wave should rather be defined in the receiver body or a receiver-carried system. Additionally, other sources of wind would also need to be modeled, such as turbulence and gusts.

The implemented module defines an abstract class `WindField` which inherits from the abstract class `WorldElement` and which defines a standard API for all wind fields. A series of classes inherits from `WindField` and implements the various types of wind fields. Some may be based on three-dimensional lookup tables of three-dimensional wind vectors, others might define turbulence fields over some areas or even the entire Earth, or define analytical 1-cosine gusts (as in the EASA CS25 / FAA Part 25 §341a discrete gust criteria). A member variable defined in the `WindField` abstract class is used to indicate its parent reference frame. The idea is that the wind field concrete (derived) classes are reference-frame-agnostic: they only work with a local coordinate system without even having to know that other

coordinate systems exist. The necessary coordinate conversions between the locations for which wind information is desired (input) and, after computing the wind in the local coordinates, the coordinate transformation to bring the wind vectors in the desired coordinate system (defined by the caller) are handled automatically at the level of the `World` class. As a consequence, programmers of concrete/derived wind field classes do not have to take care of that inside of the derived classes. The implementation of the `WindField` base class and its derived classes largely benefits from the presence of the reference frames and coordinate systems described in section III.B. Simplifying and segregating in the simulation code the wind representation and the needed coordinate transformations in the air-to-air refueling simulation programs was the starting point of the entire thought process that yields the sketch and implementation of the Simulink® extension presented in this paper.

D. Aerodynamic Model Extension for Inhomogeneous Flow

By leveraging the relative kinematics features and the modeled wind fields, it becomes fairly easy to implement various kinds of low-order aerodynamics methods (strip method, vortex lattice method, etc.). The geometry of the corresponding strip, lifting surfaces, etc. for the considered aircraft can easily be defined as series of points, lines, etc, in the defined coordinate systems. Relative wind components at the required locations and in the required local coordinates can also be obtained in a few of lines of code. The computation of the key elements required to formulate most, if not all, low-order aerodynamic problems (e.g. interaction coefficients, boundary conditions, etc.) is largely simplified. When comparing the data structures obtained when implementing such methods with the C++ module described here to the ones found in the code snippets of Ref. [17], the obtained code is much more readable and it is much easier to match the equations in the code with the mathematical equations of the corresponding method.

In the context of the air-to-air refueling simulation program, an aerodynamic model extension was created on the basis of a low-order aerodynamic method for the FMTA (Future Military Transport Aircraft) configuration. FMTA is a generic aircraft configuration that was defined between Airbus and DLR to be similar to the Airbus A400M configuration but with noticeable differences, such that the results obtained with FMTA are representative for an aircraft of this size, architecture, and configuration, but not directly comparable to those of the A400M. The aerodynamic database for the FMTA configuration has grown over approximately 15 years (based on RANS and Euler CFD computations and wind tunnel tests), such that the basis flight mechanical model is certainly much more trustworthy that the low-order aerodynamic solution, especially in terms of drag and therefore flight performance. The idea was therefore not to replace the basis aerodynamic model, but rather to use the low-order aerodynamic method to compute the effect of the inhomogeneity of the encountered wind field. Consequently, the name that was chosen for the method is *Aerodynamic Model Extension for Inhomogeneous Flow* (AMEIF).

doi: 10.2514/6.2022-0808

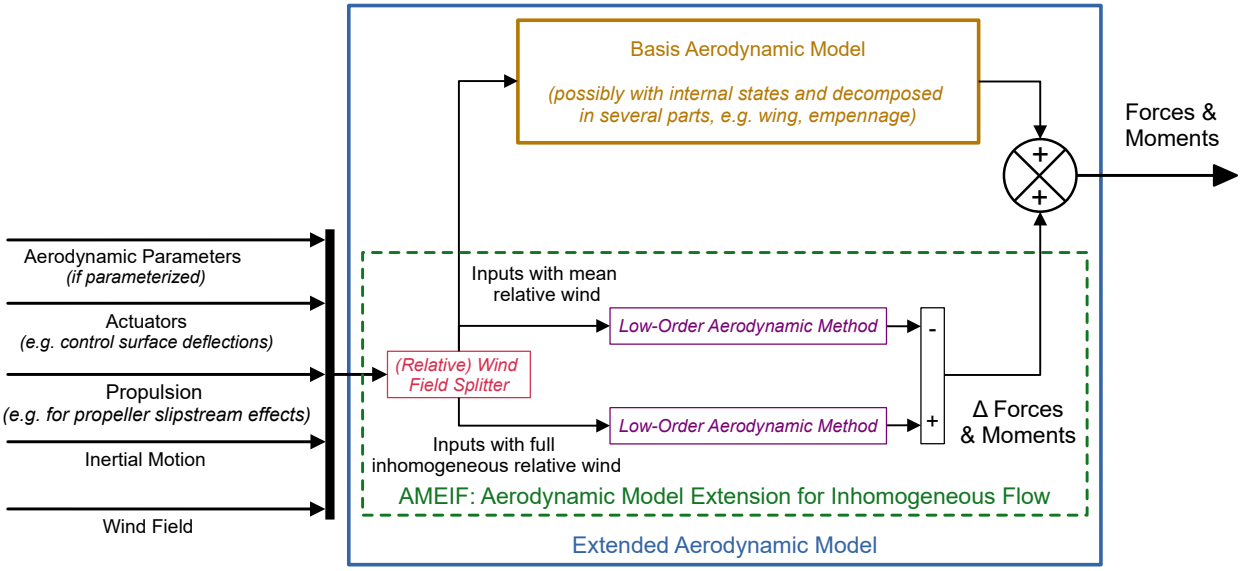


Fig. 4 Sketch of the Aerodynamic Model Extension for Inhomogeneous Flow (AMEIF)

The concept for this aerodynamic model extension is represented in Fig. 4. The basic idea is to combine the results from two low-order aerodynamic computations: one considering an average flow (same three-dimensional flow V_∞, α_∞, and β_∞ over the whole airframe) and one with the full inhomogeneous flow field. The forces and moments obtained for the homogeneous or average flow are subtracted from the forces and moments obtained for the full inhomogeneous flow field. This provides the forces and moments that are only due to the inhomogeneity of the flow. These so called Δ forces and moments are then added to the forces and moments computed with the baseline aerodynamic model and for the homogeneous/average flow. To remain consistent, it is crucial that the basis aerodynamic model is computed based on the same average flow than the first low-order aerodynamic result.

The idea is to obtain a fast method to compute the additional forces and moments due to the inhomogeneous character of the flow field that the aircraft is encountering, hence the use of low-order aerodynamic methods. The current version used with FMTA uses a numerical lifting line method, but other types of low-order aerodynamic methods could be used in the same way. In the application to the FMTA configuration, no flexible modes are modeled and the forces and moments can be reduced to a Wrench (a three-dimensional force vector, a three-dimensional moment vector, and a moment referent point). The method solves the distributed forces and moments, such that coupling this module with a flexible structural model is possible. Such coupling is planned in the near future, for applications considering flexible aircraft configurations as the High-Altitude Platform (HAP) currently being developed by DLR [18]. This coupling would for instance be useful in the framework of the piloted evaluations [19].

In terms of C++ code, the AMEIF method is integrated in a derived class, `AMEIFFrame`, of the `ReferenceFrame` class. The parts that are inherited from the `ReferenceFrame` class permit to attach the relevant aero-elements (vortex-ladder, panels, etc.) to a specific body and all computations are automatically made in the corresponding coordinate system. The matrix of aerodynamic interaction coefficients (AIC) is a member variable, with can either be fixed or be recomputed at every time step. Recomputing the AIC matrix is often not necessary, because the impact of the deformations is negligible. This feature is however crucial for highly flexible configurations with large deformations. When evaluating the flow conditions at the relevant locations, e.g. for building the right hand side of the chosen method, the programmer simply can write a couple of lines similar to the following pseudo-code:

doi: 10.2514/6.2022-0808

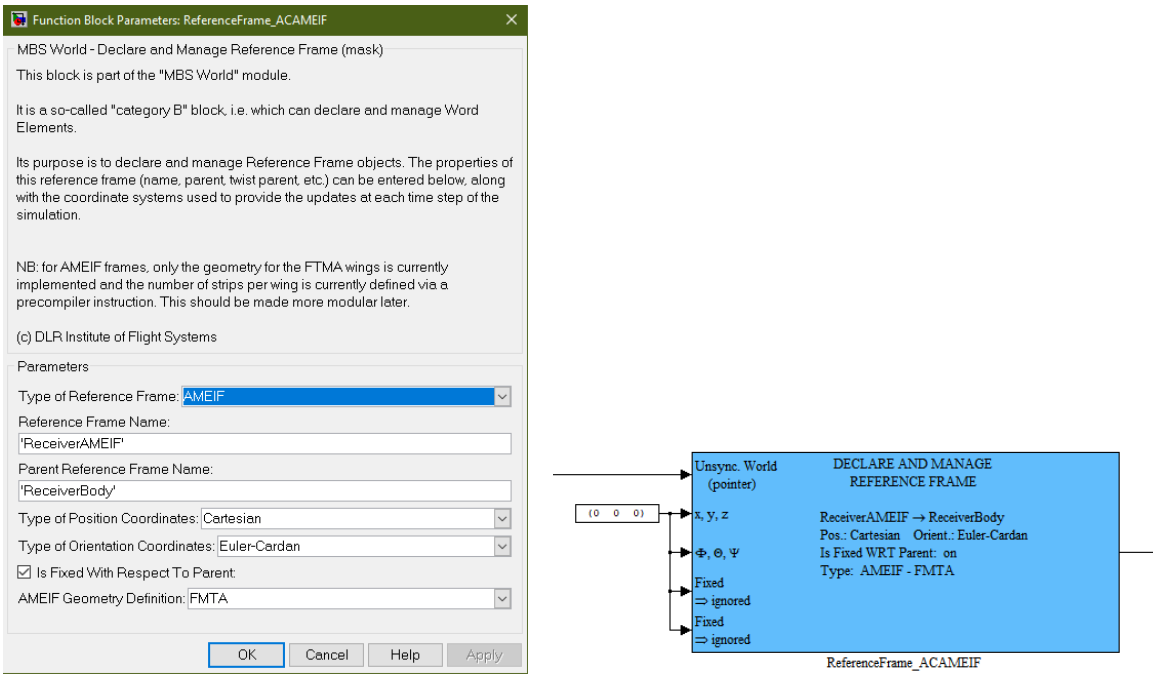
```
local_wind = my_world.get_wind_at_position( point )
local_relative_wind = local_inertial_velocity - local_wind
local_RHS_perpendicular_flow = local_relative_wind.dot_product( local_normal_vector )
```

The last two lines of this pseudo-code are somewhat trivial, provided that vector classes are providing the required functions and operators. The first line involves looking for all sources of wind and calling them for the right location in space. This involves different sets of coordinate transformations for each wind field, both for converting the input coordinates of the point of interest to each wind field's coordinate system to retrieve the corresponding wind contribution. The obtained wind vectors also need to be to be transformed from each wind field coordinate system to a common coordinate system in order to add their components. Eventually, the vector containing the sum of the different contributions is transformed to the coordinate system needed for the following computations. All this happens automatically behind the scenes: the programmer of the aerodynamic method[§§§] does not need to think about it. The same happens also behind the second and third line of this pseudo-code: the manipulated objects are not just a vector of three double precision floating point values, but carry with them the information regarding the coordinate system used. If these coordinate systems do not match, then a transformation is done automatically.

If needed, users have means at their disposal to tweak the choice of coordinate systems used in the intermediate computations, for instance to prevent unnecessary back and forth transformations. Such avoidable computations could theoretically happen, but are rare in practice and do not happen in the typical use cases. This is the reason why, it was decided to not try to cover such cases with a more clever logic as this would create an overhead for all computations, not only for the problematic cases. The current default behavior is computationally cheap and does the most efficient series of transformations in almost every practically relevant case.

The Simulink® block for declaring an AMEIF frame is the same one than for regular reference frames, which was easy to realize as the `AMEIFFrame` class inherits from the `ReferenceFrame` class. When choosing the type "AMEIF" in the configuration GUI shown in Figure 5a a new line appears at the bottom of this GUI. The user can select a specific geometry for the aerodynamic method (e.g. discretization of the wing, panels, etc.). In the example shown, the configuration for the FMTA configuration is selected. This information is reported in the middle of the Simulink® block, see Figure 5b, so that users do not need to open the GUI to check the current setting.

[§§§] or any s-function that would also need the wind information



(a) Declaration mask for the Receiver AMEIF frame

(b) Block appearance for the Receiver AMEIF frame

Fig. 5 Configuration GUI and appearance of the AMEIF frame declaration block.

doi: 10.2514/6.2022-0808

IV. Discussion

The proposed extension mechanism can be used to develop a wide-range of features in C++ instead of using block diagrams. Source code and block diagrams have different strengths and weaknesses. The features implemented by the authors and presented in this paper could all have been implemented only with standard Simulink® blocks, but that would have led to significantly higher modeling work. In this estimate made by the authors, the initial development time for the module is not considered. Accounting for the initial time investment, the development of the air-to-air refueling model in pure Simulink® would probably have been slightly faster. However, two recent further developments of the air-to-air refueling simulation model demonstrated that this Simulink® extension provides noticeable gains.

The first example consisted in integrating the wind fields with the aerodynamics of the refueling drogue and of the refueling hose and already demonstrated the usefulness of the developed extension. Until this integration, the aerodynamics model for these components received a strongly simplified wind information on their inputs. The integrated solution ensures that all sources of wind (in particular atmospheric turbulence, tanker wake, and bow-wave of the receiver) are automatically retrieved and passed as inputs to the corresponding aerodynamics models. This example is representative for the use case of a progressive transition from a block diagram implementation to a more source-code and object-oriented implementation. The main feature that was required in this case was a modified s-function allowing getting the total wind from all these wind fields at any specified point in any specified reference frame. The refueling hose is modeled as a multi-body system consisting of many hose elements that are moving with respect to the tanker, the receiver, and the Earth. The presented Simulink® extension manages the wind fields already and deals with the required coordinate transformations automatically. The *client* (i.e. category D) s-function needed could be created very rapidly and its code is very simple, as it simply manages the user configurations and mostly consists in calling the right functions from the *library*.

The second example deals with the creation of a new turbulence model, that has been specifically developed for the need of the air-to-air refueling piloted simulations. The development of the mathematical equations of this model did require significant thinking and effort, as the requirements for this turbulent model were unusual and complex. For conciseness, this aspect is not further detailed in this paper, but will certainly be published in the near future. Once the mathematical equations for that model had been designed, their implementation and integration into the overall simulation model could be performed very rapidly thanks to the capabilities of the developed Simulink® extension. This turbulence model was implemented as a new concrete derived class of the abstract class `WindField` and followed

the same API. As already mentioned in section III.C, the API of the `WindField` objects is such that the derived classes do not even have to be aware that reference frames and coordinate systems are defined in other classes; any new derived class simply has to define its own local set of coordinates and to compute everything with that set of coordinates. Eventually, the category B s-function that is used to define wind fields was extended: it now has a new type of wind field to work with and new parameters and inputs (e.g. turbulence intensity is an input that can be varied during the simulation). The complicated nature of the model with multi-dimensional turbulence fields that are implemented would have been extremly difficult, if even possible, to implement in pure Simulink®; an s-function would anyway have been the best choice. The developed framework provided all the required building blocks to implement that model in negligible time. Developing a separate s-function (i.e. not using the developed extension) would have been significantly more demanding and costly. Besides, copies of that s-function block would have had to be added manually in all places of the model where the wind information is needed. No guarantee would have been provided that the turbulence field was indeed present in all required places. Such errors cannot be made with the developed framework, which is particularly helpful when multiple variants of a model have to be maintained over long periods of time.

The authors estimate that the break-even point for this developed module was reached during the early stages of the development of these two new features. With each new development using this Simulink® extension, further gains in terms of time saved and increased flexibility are made. In addition to the productivity gains, the developers appreciate the fact that they can focus on the scientific part of the modeling process and let the module take care of the tedious and distracting coordinate transformations. Dealing with these transformations at the level of the user-defined functions would also make these functions far less readable. This was particularly noticeable during the development of the turbulence model.

Some of the source code and classes developed for this module were also already reused in completely different C++ programs of the team, which also permitted to save significant amounts of time and which would not have been the case with a Simulink® implementation. Similarly, other models and simulation programs in the authors' team have similar problems to solve and could benefit from using the developed extension or at least parts of it. The long-term savings resulting from the presented Simulink® extension can hardly be estimated at this stage, but should be very significant.

doi: 10.2514/6.2022-0808

Not all potential use-cases might be covered by the existing set of client s-functions. New s-function blocks and classes in the *World* library will certainly be needed for addressing these new use-cases. Over time though, the need for new client s-functions should decrease. Whenever the computations are generic (i.e. not too specific to that use-case), it is advisable to integrate them as new capabilities and methods in the core library and to keep the client s-funtions as simple as possible, i.e. essentially as thin interface layer between the C++ library and the MATLAB®/Simulink® environment.

V. Summary and Outlook

A Simulink® extension based on a series of s-function blocks was presented. It allows creating and managing *elements* which are C++ objects in a conceptual *world*. These objects can be put in relation with each other and, by composition, complex object structures can be built. This permits to collaboratively share information in a structured and abstract way between parts of the Simulink® model. The C++ components and classes can also provide advanced functionalities which can be accessed/called through the current or other s-function blocks. This eases the implementation of some complex simulation models by lowering the bar of using C++ s-functions for advanced features. Many advanced features become very easy to implement thanks to all the quantity of information known by the *world* object.

Adding, removing, or reconfiguring elements can usually be done by simply adding, removing, or entering different parameters in the mask of the provided Simulink® blocks (no complex configuration file and no need for recompiling the s-functions). The core library provides a good framework and set of classes and algorithms for implementing new features with or without new types of objects.

This Simulink® extension has mostly been developed in 2017 for simplifying the authors' air-to-air refueling simulation programs and is used since. It was developed as a C++98 compliant code and mostly in combination with fairly old versions of MATLAB®/Simulink® (between R2007b and R2011b). New features from newer MATLAB®/Simulink® version and C++ standards are not expected to pose any issue, but could potentially provide alternative and easier ways to implement some of the features.

The proposed extension is based on a general-purpose simulation mechanism that can easily be ported to other signal-based simulation tools (e.g. Scilab/Xcos). An implementation as a functional mock-up interface is not planed at the moment, but would be fairly straightforward. The development is mainly driven by the short-term needs of the projects of the authors' research group, such that a long-term development roadmap currently does not exist. Features

which could be developed and integrated in the near future include:

- Camera models.
- World magnetic models (WMM).
- Ground and elevation models / radio-altimeter / lidar / precised relief on runways (incl. landing gear inverse kinematics) for better immersion during on-ground maneuvering in the DLR AVES simulator.
- Relative position/bearing of the sun for a given position, date and time (e.g. to know whether a camera might be blended or not).
- highly representative simulation models of navigation aids (ILS, VOR/DME, TACAN, etc.) with real-world effects.

Appendix

A. Using RTTI to check the types of objects from different mex/s-functions under Windows

When trying to implement the type-checking mentioned in section II.D, programmers should keep in mind that the `typeid` objects obtained via the RTTI are static variables and even if comparing their addresses would work in many cases, when performing desktop simulation on the Windows platforms it would fail. In that case, each s-function is a separate dynamically linked library (dll) that is loaded by the Simulink engine. The behavior of dlls is different from those of shared libraries on Unix/Linux platforms and each dll contains a separate copy of the static variables. Consequently, the `std::type_info` objects obtained via the `typeid` operator are two distinct static variables having different addresses, whenever `typeid` is called for objects defined in different s-functions (or even different instances of the same s-function). A check would therefore always fail on Windows for non-compiled models. The problem would not occur when compiling the model as all objects would be located in the same executable and not be loaded via dlls (which the mexw32 and mexw64 compiled mex-files are).

Note also that, even if found in many code bases and examples on the internet, comparing addresses of `std::type_info` objects is officially an undefined behavior in C++, which should already be reason enough to not use address comparisons, even on non-Windows platforms.

The library and s-functions were designed to be compatible with the C++98 standard, as the real-time QNX environment and associated `qcc` compiler used in the AVES flight simulator did not support C++11 at that time. Consequently, the check was implemented based on a string comparison of the type name, as returned by the `name` member function. The `hash_code` member function of the `std::type_info` class introduced with C++11 would obviously be a better choice for any environment supporting C++11 or above. If the support for C++98 is dropped in the future, this would be changed accordingly.

doi: 10.2514/6.2022-0808

Acknowledgments

The authors would like to thank and acknowledge the colleagues involved in the DLR projects LUBETA and F(AI)^{2}R as well as those who contributed in building the DLR AVES simulator infrastructure and the FMTA model and in their everyday maintenance.

References

[1] Zipfel, P. H., "Tensor Flight Dynamics," *Proc. of the 2011 AIAA Atmospheric Flight Mechanics Conference*, Portland, OR, USA, 2011. doi: 10.2514/6.2011-6725.

[2] Zipfel, P. H., *Modeling and Simulation of Aerospace Vehicle Dynamics, 3rd ed.*, AIAA Education Series, 2014. ISBN: 978-1-62410-250-9, doi: 10.2514/4.102509.

[3] Zipfel, P. H., "A C++ Architecture for Unmanned Aerial Vehicle Simulations," *AIAA Infotech@Aerospace Conference*, Rohnert Park, CA, USA, 2007. doi: 10.2514/6.2007-2945.

[4] Deiler, C., and Fezans, N., "Virtual Flight Testing with VIRTTAC-Castor: New Capabilities and Flight Envelope Definition," *Proc. of the 2019 German Aerospace Congress (DLRK)*, Darmstadt, Germany, 2019.

[5] Fezans, N., and Deiler, C., "Inside the Virtual Test Aircraft (VIRTTAC) Benchmark Model: Simulation Architecture," *Simulation Notes Europe (SNE)*, Vol. 29, No. 1, 2019, pp. 1–12. doi: 10.11128/sne.29.on.10461.

[6] Duda, H., Gerlach, T., Advani, S., and Potter, M., "Design of the DLR AVES Research Flight Simulator," *Proc. of the 2013 AIAA Modeling and Simulation Technologies Conference*, Boston, MA, USA, 2013, pp. 1–14. doi: 10.2514/6.2013-4737, AIAA 2013-4737.

[7] Gerlach, T., and Durak, U., "AVES SDK: Bridging the Gap between Simulator and Flight Systems Designer," *Proc. of the AIAA AVIATION 2015 Modeling and Simulation Technologies Conference*, Dallas, TX, USA, 2015, pp. 1–10. doi: 10.2514/6.2015-2947, AIAA 2015-2947.

[8] Fezans, N., and Jann, T., "Modeling and Simulation for the Automation of Aerial Refueling of Military Transport Aircraft with the Probe-and-Drogue System," *Proceedings of the 2017 AIAA Aviation Forum - Modeling and Simulation Technologies*, Denver, CO, USA, 2017. doi: 10.2514/6.2017-4008.

[9] Fezans, N., and Jann, T., "Towards automation of aerial refuelling manoeuvres with the probe-and-drogue system: modelling and simulation," *Transportation Research Procedia*, Vol. 29, 2018, pp. 116–134. Proceedings of the 2017 CEAS Aerospace Europe Conference, doi: 10.1016/j.trpro.2018.02.011.

[10] Davidson, J. K., and Hunt, K. H., *Robots and Screw Theory – application of kinematics and statics to robotics*, Oxford University Press, 2004. ISBN: 978-0198562450, doi: 10.1017/CBO9780511810329.

[11] Dogan, A., Lewis, T. A., and Blake, W., "Wake-Vortex Induced Wind with Turbulence in Aerial Refueling - Part A: Flight Data Analysis," *Proc. of the 2008 AIAA Atmospheric Flight Mechanics Conference and Exhibit*, Honolulu, HI, USA, 2008, pp. 1–14. doi: 10.2514/6.2008-6696.

[12] Dogan, A., Lewis, T. A., and Blake, W., "Wake-Vortex Induced Wind with Turbulence in Aerial Refueling - Part B: Model and Simulation Validation," *Proc. of the 2008 AIAA Atmospheric Flight Mechanics Conference and Exhibit*, Honolulu, HI, USA, 2008, pp. 1–26. doi: 10.2514/6.2008-6697.

[13] Haag, C., Schwaab, M., and Blake, W., "Computational Analysis of the Bow Wave Effect in Air-to-Air Refueling," *Proc. of the 2010 AIAA Atmospheric Flight Mechanics Conference*, Toronto, ON, Canada, 2010, pp. 1–11. doi: 10.2514/6.2010-7925.

[14] Dogan, A., and Blake, W., "Modeling of Bow Wave Effect in Aerial Refueling," *Proc. of the 2010 AIAA Atmospheric Flight Mechanics Conference*, Toronto, ON, Canada, 2010, pp. 1–17. doi: 10.2514/6.2010-7926.

[15] Bhandari, U., Thomas, P. R., Bullock, S., Richardson, T. S., and du Bois, J. L., "Bow Wave Effect in Probe and Drogue Aerial Refuelling," *Proc. of the 2013 AIAA Guidance, Navigation and Control Conference*, Boston, MA, USA, 2013, pp. 1–21. doi: 10.2514/6.2013-4695.

[16] Dogan, A., Blake, W., and Haag, C., "Bow Wave Effect in Aerial Refueling: Computational Analysis and Modeling," *Journal of Aircraft*, Vol. 50, No. 6, 2013, pp. 1856–1868. doi: 10.2514/1.C032165.

[17] Katz, J., and Plotkin, A., *Low-Speed Aerodynamics, 2nd edition*, Cambridge University Press, 2001. ISBN: 978-0-521-66552-0, doi: 10.1017/CBO9780511810329.

[18] Hasan, Y. J., Roeser, M. S., Hepperle, M., Niemann, S., Voß, A., and Handojo, V., "Flight Mechanical Design and Analysis of a Solar-Powered High-Altitude Platform," *Proc. of the 2020 German Aerospace Congress (DLRK Deutscher Luft- und Raumfahrtkongress)*, Online conference, 2020.

[19] Hasan, Y. J., Roser, M. S., and Voigt, A. E., "Evaluation of the Controllability of a High-Altitude Platform in Atmospheric Disturbances Based on Pilot-in-the-Loop Simulations," *Proc. of the 2021 German Aerospace Congress (DLRK Deutscher Luft- und Raumfahrtkongress)*, Online conference, 2021.