Author version

Published as:

Backhaus, T., Gottfried, S., Merle, A., Hwang, J. T., & Stueck, A. (2021). Modularization of high-fidelity static aeroelastic MDO enabling a framework-based optimization approach for HPC. In AIAA Scitech 2021 Forum (Paper No. AIAA 2021-1236). https://doi.org/10.2514/6.2021-1236

BibTeX for this reference:

https://lsdolab.github.io/lsdo-publications/publication/backhaus2021modularization/

```
@inproceedings{backhaus2021modularization,
  author = {Thomas Backhaus and Sebastian Gottfried and Andrei Merle and John T. Hwang
    and Arthur Stueck},
  title = {Modularization of High-Fidelity Static Aeroelastic MDO Enabling a
    Framework-based Optimization Approach for HPC},
  booktitle = {AIAA Scitech 2021 Forum},
  year = {2021},
  doi = {10.2514/6.2021-1236},
  url = {https://doi.org/10.2514/6.2021-1236}
}
```

Modularization of High-Fidelity Static Aeroelastic MDO Enabling a Framework-based Optimization Approach for HPC

Thomas Backhaus*, Sebastian Gottfried†, Andrei Merle‡ and Arthur Stück§
German Aerospace Center (DLR)

John T. Hwang¶
University of California San Diego (UCSD)

A framework-based MDO approach for HPC is presented in this paper that aims at combining the modular analysis and unified derivatives (MAUD) technique implemented in OpenMDAO with the capabilities of the HPC environment FlowSimulator for multiphysics simulations. A seamless integration of OpenMDAO and FlowSimulator is suggested, which—based on a library extension—enables OpenMDAO to directly operate on central mesh data objects provided by the FlowSimulator data manager (FSDM). An existing, hand-coded gradient-based procedure for the static aeroelastic optimization of a generic transport aircraft is reorganized to be realized by the suggested framework-based MDO approach. We discuss how the existing high-fidelity analysis codes established in industry—including the viscous CFD solver TAU, a finite-element structural model, an elasticity-based volume mesh deformation, and the FlowSimulator platform—are integrated into the framework-based MDO in a modular way. This paper is a direct continuation of previous work and demonstrates how a more granular modularization of an advanced high-fidelity aeroelastic analysis and its adjoint-based gradient computation can be achieved.

Acronyms

CAD	Computer-Aided Design
CFD	Computational Fluid Dynamics
CSM	Computational Structure Mechanics
FEM	Finite Element Method
FSDM	FlowSimulator Data Manager
HPC	High Performance Computing
HTP	Horizontal Tail Plane
LNBGS	Linear Block Gauss–Seidel Solver
MAUD	Modular Analysis and Unified Derivative
MDA	Multidisciplinary Design Analysis
MDO	Multidisciplinary Design Optimization
MLS	Moving Least Square
NLBGS	Nonlinear Block Gauss–Seidel Solver
RANS	Reynolds-Averaged Navier–Stokes
ROM	Reduced-Order Model
TAU	Unstructured CFD solver developed at DLR

I. Introduction

Ever-increasing model complexities and accuracy requirements in the multidisciplinary design optimization (MDO) of detailed aircraft configurations call for scalable software solutions. Particularly in conjunction with state-of-the-art

*Research Scientist, DLR Institute of Software Methods for Product Virtualization, Department Simulation Frameworks, Zwickauer Str. 46, D-01069 Dresden, Germany.
†Research Scientist, DLR Institute of Software Methods for Product Virtualization, Department Simulation Frameworks, Zwickauer Str. 46, D-01069 Dresden, Germany.
‡Research Scientist, DLR Institute of Aerodynamics and Flow Technology, Department C²A²S²E, Lilienthalplatz 7, D-38102 Braunschweig, Germany.
§Head of Department, DLR Institute of Software Methods for Product Virtualization, Department Simulation Frameworks, Zwickauer Str. 46, D-01069 Dresden, Germany.
¶Assistant Professor, University of California San Diego, Department of Mechanical and Aerospace Engineering, 9500 Gilman Drive, San Diego, CA, USA

high-fidelity analyses such as viscous CFD or detailed CSM computations, high performance computing (HPC) integration is a key enabler. Performance gains of modern and future HPC architectures must be leveraged to increase the simulation accuracy in optimization. As future HPC performance gains are expected to be achieved by a rapidly growing number of cores per node, the software design for simulation-based optimization in multiple physics must be capable and flexible enough to allow for massively parallel computations. By means of gradient-based optimization algorithms in conjunction with an efficient computation of sensitivity derivatives, it becomes possible to tackle large optimization problems evaluated by cost-intensive numerics. The discrete-adjoint approach plays a central role in this context. Particularly for coupled multiphysics problems, the manual setup of gradient computations in adjoint mode is often both cumbersome and error-prone. Usability and automation are key ingredients to raise acceptance and usage in the context of industrial aircraft design optimization.

These challenges are addressed in this paper by combining the HPC capabilities of the ecosystem FlowSimulator for multiphysics simulations with the framework OpenMDAO [1] providing advanced algorithms for both the solution and the gradient-based optimization of multidisciplinary problems. FlowSimulator is a software infrastructure for coupled multiphysics analyses with a focus on large-scale HPC [2]. The software ecosystem FlowSimulator is a joint development by AIRBUS, ONERA, DLR and other contributors. It is an established toolset in the aeronautic industry and used in conjunction with different disciplinary tools. The FlowSimulator ecosystem provides a number of industry-proven analysis components referred to as plug-ins. Huge mesh-based simulation data sets are exchanged in fully domain-parallel workflows in memory via the central FlowSimulator Data Manager (FSDM). Special attention is on the *modular analysis and unified derivatives* (MAUD) [3] implemented in OpenMDAO. It allows for a formalized and facilitated semi-analytic derivative computation for coupled multidisciplinary problems. In addition, OpenMDAO offers a number of advanced solution algorithms on the multidisciplinary level, which makes it unique among other MDAO frameworks and a natural choice for this study.

The targeted synthesis of OpenMDAO and FlowSimulator is applied to a gradient-based aeroelastic optimization of a generic transport aircraft in trimmed equilibrium flight conditions, cf. Section II, for which a hand-coded baseline procedure exists [4–6]. The baseline procedure implemented in the FlowSimulator environment is based on high-fidelity accurate simulation methods involving finite-element structure mechanics and viscous aerodynamic CFD analyses; the latter are carried out on a sequence of refined meshes ranging from 500 thousand to 10 million mesh nodes. Applying the combined FlowSimulator-OpenMDAO approach to this high-fidelity MDO scenario we aim at significantly facilitating and accelerating the reverse-mode sensitivity analysis, while the HPC strategy of the FlowSimulator ecosystem is retained, which is domain-parallel from end to end.

The envisioned infrastructure is depicted in Figure 1 with a focus on the data organization. The FlowSimulator ecosystem is enclosed by a dashed line. Within that ecosystem, a number of simulation plug-ins are contained in the shaded area on the right-hand side. These are complemented by the gray comb located in the middle, which is centered around the FSDM. The latter centrally provides domain-decomposed mesh data for the entire ecosystem. Moreover, elementary mesh operations and services such as mesh interpolation, import/export filters and geometric transformations are offered in a central position so that they need to be implemented only once for large multidisciplinary systems. The arrows between the plug-ins and the FSDM indicate that the plug-ins either use views (glasses) on the FSDM or take local data copies (data bucket) stored in dedicated, tailored data structures of the plug-ins. An example for a simulation plug-in operating in place on FSDM data is the elasticity-analogy mesh-deformation tool FSMeshDeformation, cf. Figure 1. The finite-element structure mechanics method NASTRAN or the CFD code TAU copy the data to specialized node- or face-based data structures that are specific to the discipline. However, as the individual mesh data sets are shared in memory via the central FSDM in a domain-decomposed way and processed by the simulation plug-ins in the same decomposition throughout the entire FlowSimulator workflow, deep-copy operations can be carried out process-wise and do not lead to major efficiency losses. The data-link between OpenMDAO located on the top left of Figure 1 and the FlowSimulator ecosystem is realized through the central FSDM. We target an architecture, in which the FSDM-based strategy for sharing HPC data over the entire FlowSimulator ecosystem is extended to include OpenMDAO and supports its algorithms in a scalable way.

In this paper, we particularly focus on two major development activities that were identified to achieve the FlowSimulator-OpenMDAO integration:

- Firstly, the individual FlowSimulator plug-ins need to be modularized to be linked to the OpenMDAO framework in a fine-grained way. More precisely, the individual simulation components—i. e. FlowSimulator plug-ins—need to modularly implement the API of OpenMDAO to facilitate the sensitivity analysis by means of the MAUD implementation in OpenMDAO. Whereas huge parts of the gradient accumulation according to the chain and product rule of differentiation were hand-coded in the baseline MDO procedure,

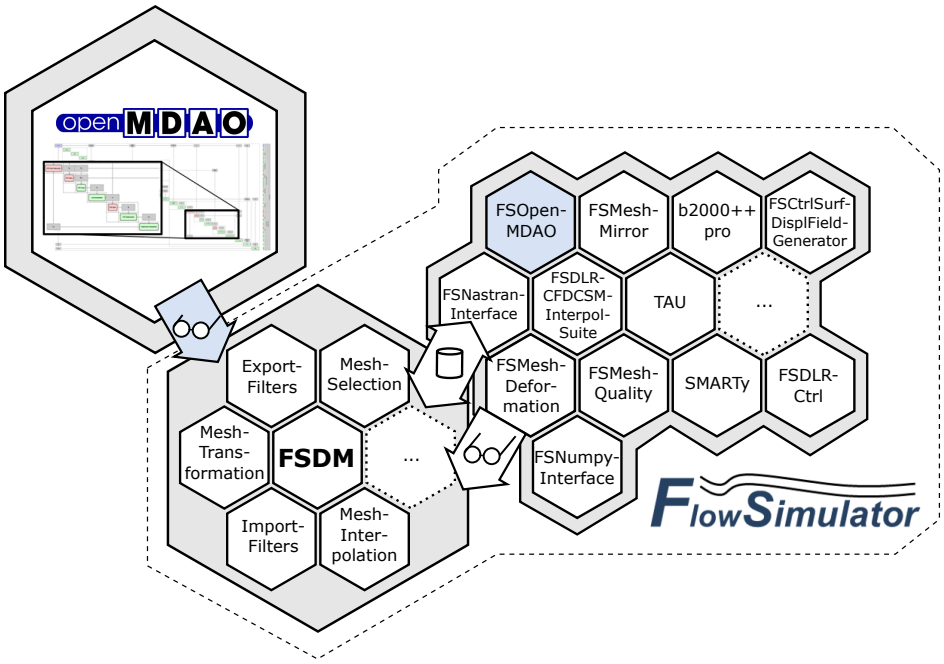

Figure 1 FlowSimulator ecosystem and OpenMDAO

the formerly plug-in-internal term assembly can thus be assigned to OpenMDAO. To efficiently process large CFD cases in parallel, a matrix-free implementation is used so that computationally intensive matrix-vector and vector-matrix products are processed within the disciplinary components.

- The second challenge that we consider crucial for a seamless integration of the FlowSimulator ecosystem and OpenMDAO is related to the data management strategy: we aim at bridging the gap between (a) mesh data neutrally provided to the FlowSimulator ecosystem by the FSDM and (b) the concept of concatenated multidisciplinary arrays held in OpenMDAO to support advanced MDAO algorithms such as MAUD. To this end, a library extension to OpenMDAO is suggested based on an abstraction of its vector interface, which enables OpenMDAO to operate in place on FSDM mesh data structures. The functionality is indicated by the blue arrow in Figure 1 and implemented in the plug-in library FSOpenMDAO.

An overview of the gradient-based MDO problem considered here and the simulation components involved is given in Section II. Subsequently, a description of the framework-based MDO method is presented in Section III. Special attention is on the software architecture and the integration of the HPC ecosystem FlowSimulator and OpenMDAO. In Section IV, we describe the modularization of the baseline MDO process to support an automated reverse-mode gradient computation by the OpenMDAO framework. The application to the considered aeroelastic MDO problem is shown in Section V including run-time comparisons for the suggested framework-based approach with regard to the evaluation of objective functionals and gradients.

II. MDO Problem and Components

The considered MDO problem of an aeroelastic aircraft configuration in trimmed flight conditions [4, 5] and the analysis tools involved are outlined below.

A. MDO Problem

The optimization considered below is built upon an existing aeroelastic coupling routine defined by Merle et al. [4, 5] According to Figure 2 an initial surface shape is defined through a set of shape parameters. Subsequently, an iterative static aeroelastic iteration cycle is carried out to couple aerodynamics and structure mechanics during the aeroelastic optimization of the aircraft by means of a nonlinear Gauss–Seidel method. Within each iteration of the cycle, changes of the aircraft surface geometry stemming from both changes in the shape parameters and the elastic deformation are propagated into the CFD volume mesh according to an elasticity-analogy mesh deformation approach.

Then, the loads on the surface of the aircraft are predicted by the CFD method and transferred to the CSM mesh. The CSM method subsequently computes the elastic deformation of the aircraft structure, which is interpolated back onto aerodynamic surface mesh, before the volume mesh is deformed once more by the mesh deformation method. Once the aeroelastic coupling is converged, the aerodynamic lift, drag and pitching coefficients are augmented by the gravitational and thrust forces to obtain the trim residuals or defects. The trimming can be addressed either using trim-corrected gradients in an unconstrained optimization method or the trim constraints can be explicitly treated by the optimizer to achieve a trimmed flight equilibrium; both techniques are studied by Merle et al. [7] for this MDO scenario. The individual components for this procedure are summarized below:

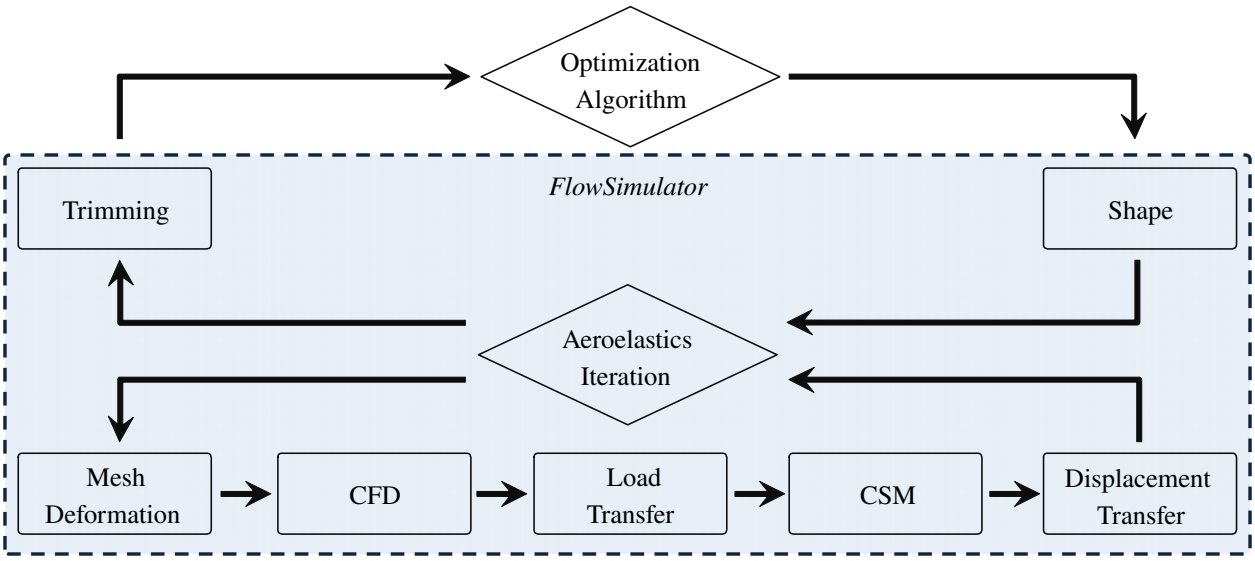

Figure 2 Aeroelastic coupling process

B. MDO Components

The following software methods—or plug-ins in the FlowSimulator ecosystem—are used in the described aeroelastic coupling group:

Shape Definition The aircraft's wing shape is parameterized using Bspline control points introducing smooth deformations on seven span-wise distributed wing sections. Each section is controlled by nine vertically moving points on each upper and lower wing side. The position of trailing and leading edges of the sections are fixed, prohibiting a change in the wings twist distribution. The total magnitude of possible shape changes is relatively small with this parameterization. This is intended, since it accounts for the fact, that the shape of the structural aircraft model is fixed during the optimization, meaning that the wing box still has to fit the outer aerodynamic shape. By modifying the wing, the intersections with other aircraft parts, i. e. body and pylon have to remain valid. This was solved by implementing the parameterization within a CAD system, in this case CATIA v5. In order to streamline the workflow on a Linux HPC cluster and to obtain analytic geometric sensitivities, a reduced-order model (ROM) representation of the parametric CAD model, see Bobrowski et al.[8] and Merle et al.[4], was built using DLR's Surrogate Modeling for AeRoData Toolbox SMARTy. The same ROM-based procedure is applied to parameterize the Horizontal Tail Plane (HTP) deflection required for trimming.

Mesh Deformation For each aerostructural coupling iteration, the current deformation fields of the wing shape parameterization, the HTP trimming rotation and the structural model deflection are summed up on the CFD surface mesh. The propagation of all surface displacements into the volume mesh is performed by means of the linear elasticity analogy (EA) implemented in DLR's mesh deformation FlowSimulator plug-in FSMeshDeformation. Therein, the linear elasticity equations are discretized based on the baseline jig shape geometry by a Finite Element Method (FEM) and solved using a block-SOR-preconditionned BiGCStab PETSc [9] solver. Since the linear EA problem is self-adjoint, the implementation of the adjoint capability in FSMeshDeformation is straightforward.

CFD The DLR TAU code [10] is a general-purpose CFD code based on a second-order accurate finite-volume discretization for compressible flow. The TAU code is well-established in academia and in daily production use in the

European aeronautics industry. It operates on fully unstructured, median-dual grids. In this MDO-framework study a steady-state Reynolds-averaged Navier–Stokes (RANS) approach was chosen in conjunction with the negative extension of the one-equation Spalart–Allmaras turbulence model (SA-neg). A central convection scheme stabilized by a matrix-valued dissipation operator was applied here. Viscous fluxes are computed from a central scheme in conjunction with a Green–Gauss gradient approximation. We used a non-linear agglomerated multigrid method according to the Full Approximation Scheme (FAS) and a smoother based on an implicit pseudo-time stepping of backward-Euler type. Inside the implicit pseudo-time stepping loop, a lower-upper Gauss–Seidel approach (LUSGS) is applied performing a forward-backward sweep with the LUSGS operator. The discrete-adjoint complement of the TAU code provides a complete differentiation of the second-order accurate finite-volume residual and a default set of standard objective functionals [11]. The explicit storage of Jacobian matrices with extended discretization stencils is avoided in the discrete-adjoint method. Instead, the adjoint residual is evaluated on-the-fly in multple sweeps [12] so that the memory requirements of the discrete-adjoint method are comparable to the nonlinear solver. The adjoint equation systems are solved by a Krylov-GMRES approach, preconditioned by a linear version of the nonlinear multigrid method above which, in turn, is iterated by a relaxed LUSGS method. It is the reverse of the LUSGS method applied to the nonlinear problem consistently using the transposed operators. The TAU code is integrated into the FlowSimulator ecosystem; it is wrapped to Python and proves an in-memory link to the FSDM mesh data structures.

CSM The initial structural linear FEM model was generated with DLR's structural design process cpacs-MONA [13]. The FEM model is discretized and solved with MSC Nastran SOL101 for the aerostructural coupling. The model consists mainly of PSHELL type shell elements in conjunction with rigid-body type RBE2 and RBE3 elements. The linear FEM problem is self-adjoint, which means that MSC Nastran SOL101 can be used with the same structural model in both the primal and the adjoint mode by just exchanging the applied aerodynamic loads, but it is crucial that constant loads, e. g. payload, fuel, systems, are deactivated in the adjoint mode.

Displacement transfer For transferring the structural displacements obtained by the FEM analysis onto the CFD mesh, an interpolation matrix based on the Moving Least Squares (MLS) algorithm [14] is used. The matrix is set-up during the first optimization iteration using the CFD and CSM baseline jig-shape meshes. Since it is constant throughout the optimization run, the matrix can be treated also as a constant in the derivation of the aerostructural coupled adjoint system. An extension of this simplified approach to update the interpolation matrix based on the current flight shapes is targeted in future work.

Load transfer Node-wise stored local aerodynamic force vectors are obtained on the CFD surface mesh by integrating pressure and friction stresses over each single CFD surface patch. The force distribution is interpolated from the CFD surface mesh onto the CSM mesh using the transpose of the MLS matrix. This imposes a consistent and conservative load and displacement transfer between the CFD surface mesh and the CSM mesh.

Trim constraint handling An implicit trimming approach is followed by trimming each design candidate suggested by the optimization algorithm. Therein, a Newton-based algorithm employing Broyden's bad method for the trim Jacobian update adjusts the aerodynamic angle of attack α, the deflection of the HTP δ and the length scale τ_e of a reference vector representing the engine thrust. The trimming target is to obtain an equilibrium of all forces and moments acting on the aircraft during the currently analyzed cruise point. Aerodynamic loads, gravitational forces and thrust forces are accounted for. The advantage of this procedure is that each design candidate is feasible regarding the trim conditions by the end of an evaluation sequence. However, the gradient of the goal functional of the optimization needs to account for a trim defect introduced by a shape parameter variation. The required modification to the goal functional gradient can be performed in the last step of the sensitivity computation and does not affect the aerostructural coupled adjoint system. The method is explained in more detail in Merle et al. [7].

C. Governing Multidisciplinary Equation System

Nonlinear Design Evaluation As mentioned above, the aeroelastic analysis procedure consists of a number of disciplinary components, which contribute to the coupled set of nonlinear, discrete equations governing the MDO

problem. In residual form, the individual component equations can be written as:

$$\text{Shape model:} \quad \mathbf{R}_{\tilde{\mathbf{U}}_\mathrm{F}}\left(\mathbf{D}, \tilde{\mathbf{U}}_\mathrm{F}\right) = \mathbf{0} \tag{1}$$

$$\text{CFD Mesh Deformation:} \quad \mathbf{R}_\mathbf{X}\left(\tilde{\mathbf{U}}_\mathrm{F}, \mathbf{X}\right) = \mathbf{0} \tag{2}$$

$$\text{CFD Solver:} \quad \mathbf{R}_\mathbf{W}(\mathbf{X}, \mathbf{W}) = \mathbf{0} \tag{3}$$

$$\text{CFD Loads:} \quad \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}(\mathbf{X}, \mathbf{W}, \boldsymbol{\tau}_\mathrm{F}) = \mathbf{0} \tag{4}$$

$$\text{CFD Loads Transfer:} \quad \mathbf{R}_{\boldsymbol{\tau}_\mathrm{S}}(\boldsymbol{\tau}_\mathrm{F}, \boldsymbol{\tau}_\mathrm{S}) = \mathbf{0} \tag{5}$$

$$\text{CSM Solver:} \quad \mathbf{R}_\mathbf{Q}(\boldsymbol{\tau}_\mathrm{S}, \mathbf{Q}) = \mathbf{0} \tag{6}$$

$$\text{CSM Displacements Extraction:} \quad \mathbf{R}_{\mathbf{U}_\mathrm{S}}(\mathbf{Q}, \mathbf{U}_\mathrm{S}) = \mathbf{0} \tag{7}$$

$$\text{Displacement Transfer:} \quad \mathbf{R}_{\mathbf{U}_\mathrm{F}}(\mathbf{U}_S, \mathbf{U}_\mathrm{F}) = \mathbf{0} \tag{8}$$

The residual of the CFD mesh deformation component $\mathbf{R}_\mathbf{X}$ depends on the surface mesh displacement $\tilde{\mathbf{U}}_\mathrm{F}$ applied to the surface mesh and the nodal coordinates of the CFD volume mesh, $\mathbf{X}$. $\tilde{\mathbf{U}}_\mathrm{F}$ is either fed from the shape definition module or from the structural deformation interpolated to the CFD surface mesh within the aeroelastic coupling loop. The CFD mesh deformation solver propagates the displacements applied to the CFD surface mesh coordinates $\tilde{\mathbf{U}}_F$ to the nodal coordinates of the CFD volume mesh, $\mathbf{X}$. Note that the evolving deformations are always applied to the baseline mesh. The residual of the *CFD Solver* component $\mathbf{R}_\mathbf{W}$ depends on the CFD volume mesh $\mathbf{X}$ and the CFD states $\mathbf{W}$. It is followed by the *CFD Loads* evaluation component that outputs the aerodynamic surfaces stresses $\boldsymbol{\tau}_\mathrm{F}$. The *Load Transfer* component transfers the loads of the CFD surface mesh to the surface of the CSM mesh $\boldsymbol{\tau}_\mathrm{S}$. The residual of the CSM solver, $\mathbf{R}_\mathbf{Q}$, depends on the aerodynamic loads on the CSM mesh and the CSM state variables $\mathbf{Q}$ computed by the CSM solver. Subsequently, the CSM displacements $\mathbf{U}_S$ are extracted from the CSM states, which are then transferred back on the CFD mesh by the *Displacement Transfer* component.

Adjoint-based Gradient Computation A coupled system of equations is obtained by collecting the individual residual vectors into a global residual vector $\mathbf{R}_{\boldsymbol{\Phi}}^\mathrm{T} = [\mathbf{R}_{\tilde{\mathbf{U}}_\mathrm{F}}^\mathrm{T}, \mathbf{R}_\mathbf{X}^\mathrm{T}, \mathbf{R}_\mathbf{W}^\mathrm{T}, \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}^\mathrm{T}, \mathbf{R}_{\boldsymbol{\tau}_\mathrm{S}}^\mathrm{T}, \mathbf{R}_\mathbf{Q}^\mathrm{T}, \mathbf{R}_{\mathbf{U}_\mathrm{S}}^\mathrm{T}, \mathbf{R}_{\mathbf{U}_\mathrm{F}}^\mathrm{T}]$. Accordingly, a global state vector $\boldsymbol{\Phi}$ is defined, consisting of the individual states governed by the residual equations. All residual and state vectors introduced above are assumed to be column vectors. For the subsequent Jacobian matrices, the differentiation indices are running row-wise. The total derivative of the scalar objective functional J can either be obtained in direct mode via

$$\frac{\mathrm{d}J}{\mathrm{d}\mathbf{D}} = \frac{\partial J}{\partial \boldsymbol{\Phi}} \frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}\mathbf{D}} \qquad \text{with} \qquad \frac{\partial \mathbf{R}_{\boldsymbol{\Phi}}}{\partial \boldsymbol{\Phi}} \frac{\mathrm{d}\boldsymbol{\Phi}}{\mathrm{d}\mathbf{D}} = -\frac{\partial \mathbf{R}_{\boldsymbol{\Phi}}}{\partial \mathbf{D}} \tag{9}$$

or in adjoint mode via

$$\frac{\mathrm{d}J}{\mathrm{d}\mathbf{D}} = (\boldsymbol{\Phi}^*)^\mathrm{T} \frac{\partial \mathbf{R}_{\boldsymbol{\Phi}}}{\partial \mathbf{D}} \qquad \text{with} \qquad \left(\frac{\partial \mathbf{R}_{\boldsymbol{\Phi}}}{\partial \boldsymbol{\Phi}}\right)^\mathrm{T} \boldsymbol{\Phi}^* = -\left(\frac{\partial J}{\partial \boldsymbol{\Phi}}\right)^\mathrm{T} . \tag{10}$$

As J is not an explicit function of $\mathbf{D}$ in our case, the corresponding partial derivative is neglected here for brevity. Recall that in direct mode, the right-hand side equation system (9) needs to be solved n times with n being the dimensionality of $\mathbf{D}$ in order to obtain $\mathrm{d}\boldsymbol{\Phi}/\mathrm{d}\mathbf{D}$. In the adjoint approach, on the contrary, the right-hand side equation system (10) needs to be solved only once given the objective functional J is a scalar expression. Since we are dealing with a scalar objective functional and a few constraints, the adjoint aproach (10) is pursued here. According to Eqn. (10), the expansion of the coupled adjoint equation system for the aeroelastics group (trimming not considered here) reads

$$
\begin{pmatrix}
\frac{\partial \mathbf{R}_{\tilde{\mathbf{U}}_\mathrm{F}}}{\partial \tilde{\mathbf{U}}_\mathrm{F}}^\mathrm{T} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{\partial \mathbf{R}_\mathbf{X}}{\partial \mathbf{X}}^\mathrm{T} & \frac{\partial \mathbf{R}_\mathbf{W}}{\partial \mathbf{X}}^\mathrm{T} & \frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{X}}^\mathrm{T} & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{\partial \mathbf{R}_\mathbf{W}}{\partial \mathbf{W}}^\mathrm{T} & \frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{W}}^\mathrm{T} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \boldsymbol{\tau}_\mathrm{F}}^\mathrm{T} & \frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{S}}}{\partial \boldsymbol{\tau}_\mathrm{F}}^\mathrm{T} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{S}}}{\partial \boldsymbol{\tau}_\mathrm{S}}^\mathrm{T} & \frac{\partial \mathbf{R}_\mathbf{Q}}{\partial \boldsymbol{\tau}_\mathrm{S}}^\mathrm{T} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{\partial \mathbf{R}_\mathbf{Q}}{\partial \mathbf{Q}}^\mathrm{T} & \frac{\partial \mathbf{R}_{\mathbf{U}_\mathrm{S}}}{\partial \mathbf{Q}}^\mathrm{T} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial \mathbf{R}_{\mathbf{U}_\mathrm{S}}}{\partial \mathbf{U}_\mathrm{S}}^\mathrm{T} & \frac{\partial \mathbf{R}_{\mathbf{U}_\mathrm{F}}}{\partial \mathbf{U}_\mathrm{S}}^\mathrm{T} \\
\frac{\partial \mathbf{R}_{\tilde{\mathbf{U}}_\mathrm{F}}}{\partial \mathbf{U}_\mathrm{F}}^\mathrm{T} & \frac{\partial \mathbf{R}_\mathbf{X}}{\partial \mathbf{U}_\mathrm{F}}^\mathrm{T} & 0 & 0 & 0 & 0 & 0 & \frac{\partial \mathbf{R}_{\mathbf{U}_\mathrm{F}}}{\partial \mathbf{U}_\mathrm{F}}^\mathrm{T}
\end{pmatrix}
\begin{pmatrix}
\tilde{\mathbf{U}}_\mathrm{F}^* \\ \mathbf{X}^* \\ \mathbf{W}^* \\ \boldsymbol{\tau}_\mathrm{F}^* \\ \boldsymbol{\tau}_\mathrm{S}^* \\ \mathbf{Q}^* \\ \mathbf{U}_\mathrm{S}^* \\ \mathbf{U}_\mathrm{F}^*
\end{pmatrix}
=
-\begin{pmatrix}
\frac{\partial J}{\partial \tilde{\mathbf{U}}_\mathrm{F}} \\ \frac{\partial J}{\partial \mathbf{X}} \\ \frac{\partial J}{\partial \mathbf{W}} \\ \frac{\partial J}{\partial \boldsymbol{\tau}_\mathrm{F}} \\ \frac{\partial J}{\partial \boldsymbol{\tau}_\mathrm{S}} \\ \frac{\partial J}{\partial \mathbf{Q}} \\ \frac{\partial J}{\partial \mathbf{U}_\mathrm{S}} \\ \frac{\partial J}{\partial \mathbf{U}_\mathrm{F}}
\end{pmatrix}. \tag{11}
$$

When the coupled equation system is considered as a block-Gauss–Seidel problem with the main-diagonal contributions of Eqn. (11) on the left-hand side and the off-diagonal contributions added to the right-hand side, the block rows of the adjoint mesh deformation equation and the adjoint CFD code, respectively, can be written as

$$
\text{CFD Mesh Deformation:} \quad \left(\frac{\partial \mathbf{R}_\mathbf{X}}{\partial \mathbf{X}}\right)^\mathrm{T} \mathbf{X}^* = \mathbf{RHS}_\mathbf{X}
$$

$$
\text{with} \quad \mathbf{RHS}_\mathbf{X} = -\left(\frac{\partial J}{\partial \mathbf{X}}\right)^\mathrm{T} - \left(\frac{\partial \mathbf{R}_\mathbf{W}}{\partial \mathbf{X}}\right)^\mathrm{T} \mathbf{W}^* - \left(\frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{X}}\right)^\mathrm{T} \boldsymbol{\tau}_\mathrm{F}^* \tag{12}
$$

$$
\text{CFD Solver:} \quad \left(\frac{\partial \mathbf{R}_\mathbf{W}}{\partial \mathbf{W}}\right)^\mathrm{T} \mathbf{W}^* = \mathbf{RHS}_\mathbf{W}
$$

$$
\text{with} \quad \mathbf{RHS}_\mathbf{W} = -\left(\frac{\partial J}{\partial \mathbf{W}}\right)^\mathrm{T} - \left(\frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{W}}\right)^\mathrm{T} \boldsymbol{\tau}_\mathrm{F}^*, \tag{13}
$$

with $\partial \mathbf{R}_\mathbf{X}/\partial \mathbf{X}$ being the linear stiffness matrix $\mathbf{K}$ of the mesh deformation tool introduced above. Note that in the baseline MDO process [4, 5], which is a hand-coded procedure based on the FlowSimulator environment, the right-hand side terms $\mathbf{RHS}_\mathbf{W}$ and $\mathbf{RHS}_\mathbf{X}$ are manually assembled and coded within the individual FlowSimulator plug-ins. However, parts of this implementation were reused in a previous study, where OpenMDAO was used in conjunction with the FlowSimulator ecosystem for the first time [6]. In the previous study, the right-hand side vectors pre-assembled inside the plug-ins (components) were reused by OpenMDAO, as the modular framework integration of the components targeted here was not yet available.

III. MDO Software Architecture

A. Baseline HPC Architecture based on FlowSimulator

The baseline implementation of the aeroelastic optimization process as presented by Merle et al. [5] relies on the FlowSimulator that enables multidisciplinary evaluations of objective functionals, constraints and the corresponding derivatives. As depicted in Figure 3, the HPC ecosystem FlowSimulator is structured in three layers, the first of which is the FSDM for centrally sharing mesh data between the individual simulation plug-ins. The individual simulation plug-ins contained in the process are vertically arranged in the second layer. This indicates that both direct communication and data exchange between two or more simulation plug-ins are avoided; instead, mesh data is systematically shared in memory via the FSDM. This eliminates the need for up to $n \times (n-1)$ direct plug-in-to-plug-in communications and bilateral definitions of plug-in-specific exchange data formats and individual data access APIs. The so-called control layer on the third level is Python-coded and orchestrates the workflows for the evaluation of functionals and derivatives. In the baseline implementation, dedicated control procedures are provided for the nonlinear design evaluation and the multidisciplinary derivative computation. The simulation plug-ins and the FSDM both offer Python interfaces to be called by the control layer. According to Figure 3, an external optimizer directly calls the control layer to compute the required objective functionals and the multidisciplinary sensitivity derivatives. Like the nonlinear evaluation process,

also the adjoint-mode gradient evaluation is a hand-coded procedure—in large parts hidden inside the individual plug-ins of the FlowSimulator process.

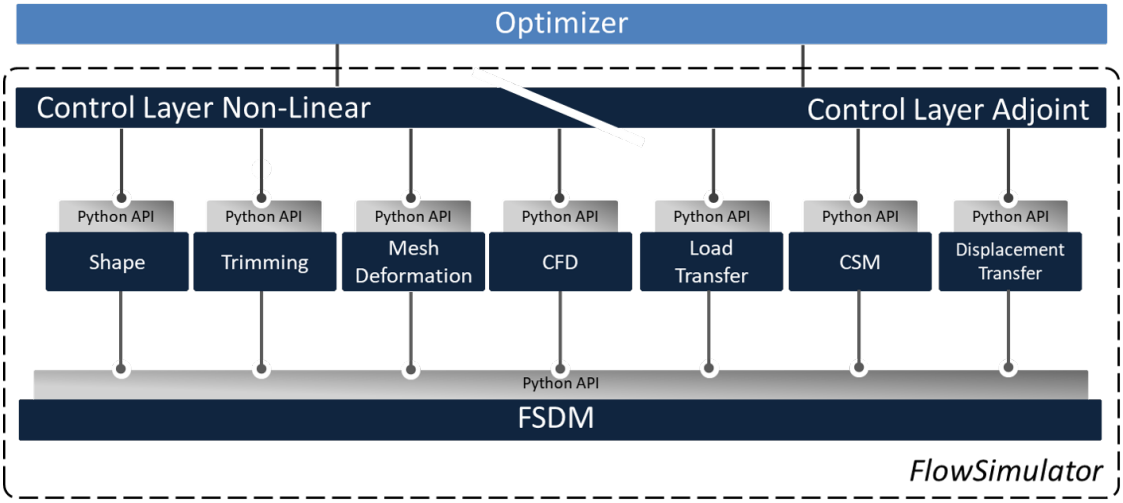

Figure 3 FlowSimulator HPC architecture for the evaluation of multidisciplinary functionals and sensitivity derivatives

B. The OpenMDAO Software Framework

OpenMDAO is an open-source software framework written in Python that facilitates the implementation and execution of large-scale MDO algorithms [1]. The development of OpenMDAO is led by the NASA Glenn Research Center. OpenMDAO is designed to enable efficient solution of large-scale optimization problems—those involving hundreds or more design variables. More specifically, it facilitates semi-analytic derivative computation methods that make it unique among MDO frameworks.

To facilitate computing derivatives, OpenMDAO implements the MAUD architecture [3]. As is the case in all modeling frameworks, the MAUD architecture assumes the model is decomposed into a set of components that define mappings from its inputs to its outputs. However, what is unique is that the MAUD architecture is based on an equation that unifies the various methods for computing the derivatives of discrete models. This equation is

$$\frac{\partial \mathcal{R}}{\partial \mathbf{u}} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{r}} = \mathcal{I} = \frac{\partial \mathcal{R}}{\partial \mathbf{u}}^{\mathrm{T}} \frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\mathbf{r}}^{\mathrm{T}}, \tag{14}$$

where $\mathbf{u} \in \mathbb{R}^n$ is a vector that concatenates all the outputs of all components in the model, and $\mathcal{R}$ is a combined residual function that constrains the entries of $\mathbf{u}$ to their appropriate values given that $\mathcal{R}(\mathbf{u}) = \mathbf{0}$ is enforced. For instance, if a variable u_i is computed by a component via a function U_i, such that $u_i = U_i(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n)$, then the corresponding residual is defined as $\mathcal{R}_i(\mathbf{u}) = u_i - U_i(u_1, \ldots, u_{i-1}, u_{i+1}, \ldots, u_n)$. Based on the structure of the model (and thus the structure of the model Jacobian, $\partial\mathcal{R}/\partial\mathbf{u}$), Eqn. (14) reduces to the chain rule, the adjoint method, or another variant or hybrid method, whichever is appropriate.

Therefore, in all situations, MAUD simplifies the task of computing the model derivatives efficiently and accurately to two steps: (1) computing the partial derivatives of each component and (2) computing the total derivatives by assembling the partial derivatives using Eqn. (14), which can be performed centrally by the framework. By adopting MAUD, OpenMDAO reduces the user effort to defining components at the appropriate level of model decomposition and computing the component derivatives.

As an example of how MAUD simplifies total derivative computation, we can show that Eqn. 14 is the basis for all of the adjoint equations in the previous section. To do this, we define

$$\mathbf{u}^{\mathrm{T}} = [\mathbf{D}^{\mathrm{T}}, \boldsymbol{\Phi}^{\mathrm{T}}, J^{\mathrm{T}}] \quad \text{and} \quad \mathcal{R}(\mathbf{u})^{\mathrm{T}} = [\mathbf{D}^{\mathrm{T}} - \mathbf{D}_0^{\mathrm{T}}, \mathbf{R}_{\boldsymbol{\Phi}}, J - J(\mathbf{D}, \boldsymbol{\Phi})], \tag{15}$$

where $\mathbf{D}_0$ is the value of the design variable vector in the current optimization iteration. Inserting these definitions into

the right equality of Eqn. 14 and back-solving to compute the final column of $(\mathrm{d}\mathbf{u}/\mathrm{d}\mathbf{r})^{\mathrm{T}}$ will yield all of the adjoint equations in the previous section.

OpenMDAO makes heavy use of object-oriented programming (OOP). The components are objects of type `Component`, whose methods including the mapping from inputs to outputs as well as the derivative computation. The `Component` instances are organized into a model hierarchy using `Group` instances that, as the name suggests, group together other groups as well as components. The `Component` class is used via one of two subclasses—`ExplicitComponent` and `ImplicitComponent`, which implement explicit and implicit functions, respectively, to define their outputs.

In the implementation of the high-fidelity static aeroelastic analysis within OpenMDAO, an important set of decisions is the choices of components and their types—`ExplicitComponent` or `ImplicitComponent`—with which to implement the model. The types of components and the dependency structure defined by the interactions between the components defines the method of derivative computation that Eqn. (14) reduces to, i. e., that OpenMDAO automatically uses. These decisions have a large effect on efficiency in terms of CPU time and memory. The sections that follow describe these decisions.

The unique aspect of our static aeroelastic optimization methodology is the implementation within an MDAO framework, OpenMDAO, systematically integrating the individual simulation plug-ins and the central mesh-data management FSDM provided by the FlowSimulator ecosystem. The use of OpenMDAO facilitates a modular model construction, which simplifies software development, and assists with assembling total derivatives via the combination of the adjoint method and the chain rule. As mentioned in the previous section, total derivatives are computed in OpenMDAO using the MAUD architecture. An existing CFD solver, ADflow, is likewise implemented in a modular way using the OpenMDAO framework [15]. To the best of the authors' knowledge, ADflow is the only CFD solver that has been adapted to enable a framework-based approach for optimization using the MAUD architecture.

However, the modular decomposition of ADflow that has been demonstrated previously has a coarser granularity than the decomposition used here. The prior work treat the entire CFD solver as a single OpenMDAO component that outputs aerodynamic coefficients [16, 17], or decompose it into (1) an implicitly-defined OpenMDAO component that outputs the CFD states given geometric design variables and (2) an explicitly-defined OpenMDAO component that outputs aerodynamic coefficients given the CFD states [15]. In contrast, the approach here uses separate OpenMDAO components for geometry parametrization, CFD mesh deformation, the implicit CFD state computation, CFD loads and interpolation, the implicit CSM state computation, CSM displacements and interpolation, and the aerodynamic coefficients. This finer decomposition increases the advantages of the modular approach, simplifying modifications to the workflow, such as replacing any of the steps above with an alternate implementation.

C. HPC Architecture based on OpenMDAO and FlowSimulator

In the baseline MDO process by Merle et al. [4, 5], Figure 3, an external optimizer triggers evaluations of cost functionals, constraints and the corresponding derivatives through dedicated FlowSimulator instances. Dedicated Python procedures are provided for both functional and derivative evaluations, which are coded separately. As opposed to the baseline procedure, the MDO problem is set up in a hierarchical manner with OpenMDAO in this study. To this end, both the external optimizer and the dedicated Python control procedures are replaced by the OpenMDAO framework, while the underlying FlowSimulator infrastructure is retained, cf. Figure 4. The parallel in-memory sharing of data between the plug-ins by means of the FSDM is extended to include OpenMDAO. The approach is discussed in detail below. According to Figure 4, the optimization problem is defined on the top level which, in turn, contains a number of subgroups, among them the trim-aeroelastic group and the aeroelastic group. Different nonlinear and linear solution strategies can be assigned to the individual, nested groups. The individual FlowSimulator components are wrapped as implicit or explicit components by implementing the Python API of OpenMDAO. This is illustrated in Figure 4 by the component wrapper boxes surrounding the individual plug-in Python APIs used in the baseline process. An example for a simplified wrapper class is presented in the subsequent paragraph, Listing 1. Thus, the data flow is fully exposed to the framework down to the component level. For example, this information is required to enable the automated gradient computation by means of the MAUD implementation provided by OpenMDAO.

D. In-place handling of parallel FlowSimulator data in OpenMDAO

With the FSDM, there is a central simulation data repository to store an arbitrary number of MPI-distributed mesh objects with associated data sets. The individual FlowSimulator plug-ins share the central FSDM to exchange data between each other, either by copying between the relevant FSDM data sets and their dedicated, internal data

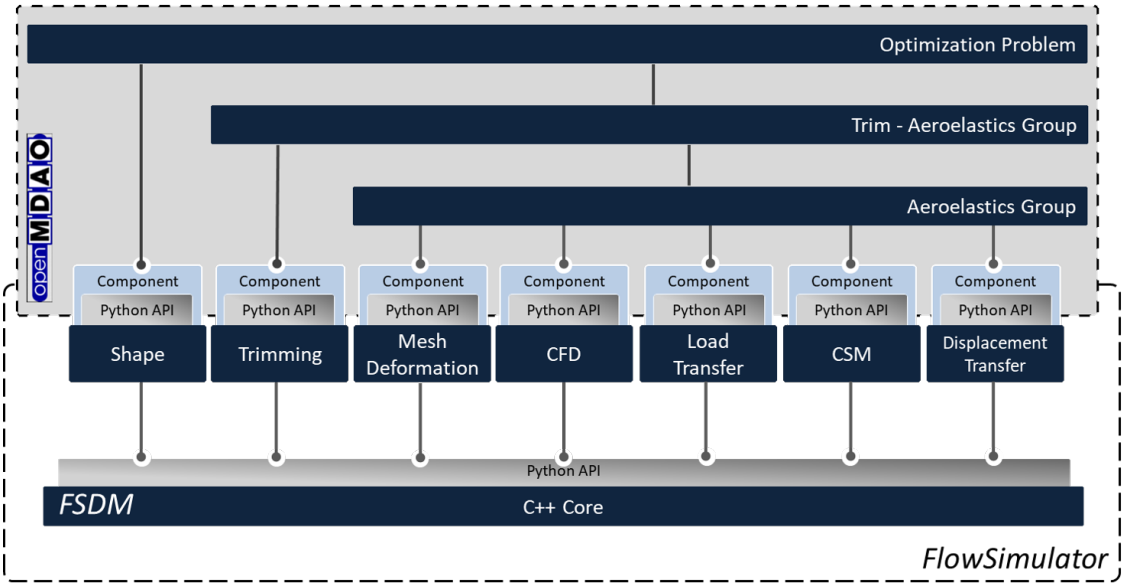

Figure 4 Framework HPC architecture based on OpenMDAO and FlowSimulator

structures (e. g. TAU) or by operating on the datatset directly (e. g. FSDLRCFDCSMInterpolSuite). In both cases, all FlowSimulator plug-ins are linked to common data centrally provided for MDAO processes. Moreover, elementary mesh operations can be offered by the FSDM itself for the common data format. To enable a central FSDM data handling for OpenMDAO-based optimization processes, a library extension to the OpenMDAO framework is suggested extending several core abstractions in order give OpenMDAO direct access to OpenMDAO data. Figure 5 gives an overview about the framework extensions.

OpenMDAO uses the `Vector` class hierarchy to store the process data. Every `System`—the base class of all component and group classes—instance keeps a set of such `Vector` objects. These vector objects are concatenated column vectors of all system variables of a given kind. OpenMDAO distinguishes between linear and nonlinear inputs, outputs, and residuals. So each system has total of six `Vector` objects to store all encapsulated variables.*

In an OpenMDAO model, all vectors of one kind build a tree-like data structure. Only the root vector of the top-level group allocates memory (realized with a NumPy `ndarray`) to store its concatenated column vector. The subsystem vectors only store a view on the slice of root vector's `ndarray` spanning the variables encapsulated by this subsystem. This allows OpenMDAO to have a unified view of the system variables on every level of the model without duplicating the data. The `Vector` class exposes this column vector with the `_data` attribute to the OpenMDAO framework. Additionally, it provides an interface for basic vector-spanning arithmetic operations and for individual variable access. OpenMDAO's driver and solver classes use this interface to access the relevant data for their algorithm implementations.

In addition to the `Vector` class hierarchy a `Transfer` class hierarchy is defined. OpenMDAO uses `Transfer` objects to communicate between components. They can either copy the data between the input-output-relationship of a pair of components or for all connections between subsystems in a group at once. Which mode is used depends on the chosen solvers for the group. Block-Gauss–Seidel solvers use pairwise transfers, block-Jacobi solvers use bulk transfer.

OpenMDAO has two vector implementations: `DefaultVector` for serial problems, and `PETScVector` for MPI-distributed parallel problems, with their matching transfer classes: `DefaultTransfer` and `PETScTransfer`. `DefaultVector` is the baseline vector implementation and operates as described above. `PETScVector` is an MPI-distributed vector implementation using PETSc `Vec` objects to share the variable data between ranks. Its transfer class enables local and cross-domain communication via PETSc's generalized scatter/gather operation `VecScatter`.

This data handling strategy is very different to the FlowSimulator ecosystem making it difficult to integrate FSDM data into OpenMDAO. In particular, there is a need for a method to represent the individual FSDM data sets as a

*OpenMDAO uses additional `Vector` objects to store e. g. variable bounds or scaling factors. They are of no concern for the in-place data handling, though, and are not discussed here.

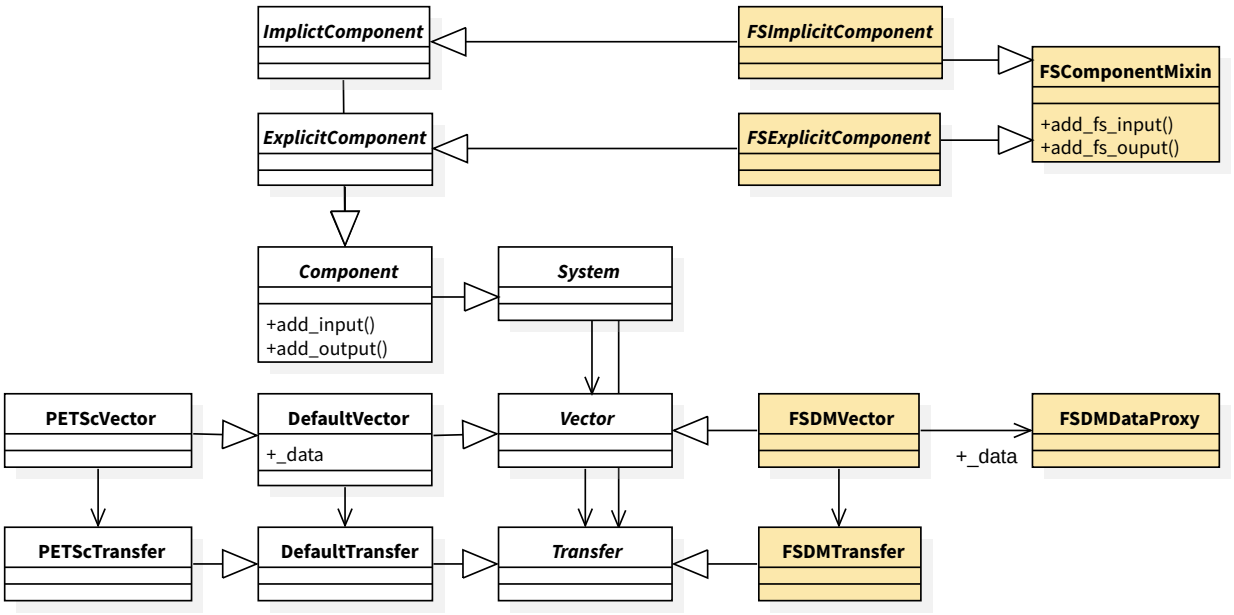

Figure 5 OpenMDAO's `Vector`, `Transfer` and `Component` class hierarchies with the newly introduced classes (highlighted in yellow) for the FSDM integration.

contiguous column vector as used in OpenMDAO. This problem is solved by the new `FSDMVector` implementation. It implements OpenMDAO's `Vector` interface but stores each OpenMDAO variable (i. e. one component input, output or residual) in an FSDM data set. With the help of `FSDMDataProxy` these data sets are contiguously assembled to one virtual column vector. This class can be used just like an `ndarray` with support for arithmetic operators, indexing, (recursive) slicing and NumPy's universal functions. When FSDMVectors are used via the vector interface, all read and write accesses to this virtual vector are effectively forwarded to the original FSDM data sets. This `FSDMVector` class is accompanied by a corresponding transfer class `FSDMTransfer`. This transfer class functions are closely related to OpenMDAO's own `PETScTransfer`, but they skip transfer operations between input and output connections sharing the same data set. Additionally it validates that each shared data set is reflected by a input/output connection by the affected components. This ensures the data flow controlled by the FlowSimulator ecosystem is in sync with the control flow defined by the OpenMDAO model.

This approach is illustrated in Figure 6. In the middle part, there is the FSDM with the exemplary data sets x, y, z, and r_z (which may be associated with a mesh object). Two simple FlowSimulator plug-ins are defined on these data sets. f computes y from its input x making it the equivalent of an explicit component in OpenMDAO. g corresponds to an implicit component computing residual r_z for the data set z. As usual for FlowSimulator plug-ins, the data set for its input y is directly shared with f. On the OpenMDAO side, the input, output, residual system vector are created with the new `FSDMVector` class, a concatenated view of arbitrary FDSM data sets. As shown with the variable y, the same data set may be mapped to more than one vector. This is important for data sets which are used by multiple components as either input or output.

To define the mappings from OpenMDAO variables to FSDM data sets, the new component base classes `FSExplicitComponent` and `FSImplicitComponent` are introduced. They implement the new `add_fs_input` and `add_fs_output` methods allowing process implementors to define FSDM-based inputs and outputs in their models. In comparison to their OpenMDAO counterparts, these methods omit the arguments for the variable shape and initial value. Instead, one can set the FSDM data sets for the variable's nonlinear and linear value and—for outputs—residual. Shape and initial value are automatically inferred from the configured data sets. These data sets are recorded in the variable metadata and are later used by the `FSDMVector` class to assemble the system vectors. See Listing 1 for an example implementation for the model as presented in Figure 6.

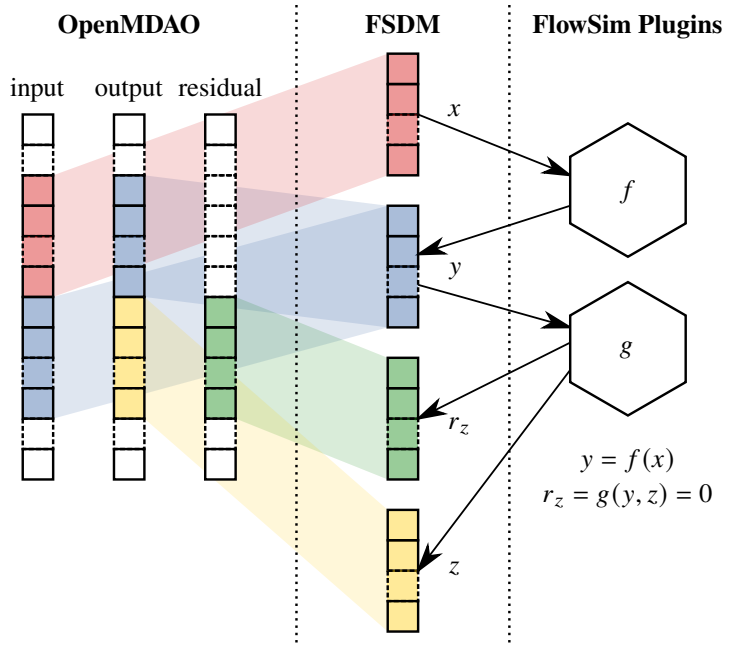

Figure 6 Data sharing between OpenMDAO and FlowSimulator using the `FSDMVector` implementation. OpenMDAO operates on views of the FSDM data sets assembled to virtual column vector proxys for the input, output and residual values.

```
class F(FSExplicitComponent):
    def setup(self):
        m = self.options['dm'].GetMesh('example')
        self.f_plugin = FPlugin(self.options['dm'])
        self.add_fs_input('x', nl_arr=m.GetUnstructDataset('x'))
        self.add_fs_output('y', nl_arr=m.GetUnstructDataset('y'))
    def compute(self, inputs, outputs):
        self.f_plugin.run_primal()  # operates exclusively on FSDM data sets

class G(FSImplicitComponent):
    def setup(self):
        m = self.options['dm'].GetMesh('example')
        self.g_plugin = GPlugin(self.options['dm'])
        self.add_fs_input('y', nl_arr=m.GetUnstructDataset('y'))
        self.add_fs_output('z', nl_arr=m.GetUnstructDataset('z'),
                                nl_res_arr=m.GetUnstructDataset('rz'))
    def apply_nonlinear(self, inputs, outputs, residuals):
        self.g_plugin.compute_residuals()
    def solve_nonlinear(self, inputs, outputs):
        self.g_plugin.run_primal()

dm = FSDataManager(FSClac(MPI.COMM_WORLD))
load_mesh_data(dm)
problem = Problem(comm=MPI.COMM_WORLD)
problem.model.add_subsystem('f', F(dm=dm))
problem.model.add_subsystem('g', G(dm=dm))
problem.setup(distributed_vector_class=FSDMVector)
problem.run_model()
print(problem['z'])  # reads value from FSDM data set
```

Listing 1 Example for an OpenMDAO model using the `FSDMVector` implementation to seamlessly integrate FlowSimulator plug-ins and data sets.

This demonstrates how, by construction, the `FSDMVector` approach guarantees consistent views for OpenMDAO on the data stored in the FSDM eliminating any redundant data between OpenMDAO and FlowSimulator. This also

removes the need to copy data between the two frameworks. `FSDMVector` operates as a seamless extension for the OpenMDAO framework requiring no changes to the framework to accommodate the new vector implementation. Existing OpenMDAO problems can be executed unmodified with `FSDMVector`. The new component base classes allow concise implementations of FlowSimulator plug-in-based components with automatic data sharing to OpenMDAO. Its performance is evaluated in Section V in conjunction with the considered MDO scenario.

IV. Framework-based Process Modularization

Overall optimization definition In order to leverage the potential of OpenMDAO's MAUD capabilities [18] the individual disciplinary components have to be reassembled as illustrated in Figure 7 and 8 in order to solve the optimization problem:

$$\min_{\mathbf{D}\in\mathbb{R}^{126},\mathbf{T}\in\mathbb{R}^{3}} \quad J\,(\mathbf{D},\mathbf{T}) \qquad \text{subject to} \qquad \mathbf{E}(\mathbf{D},\mathbf{T}) = 0\,. \tag{16}$$

The design vector $\mathbf{D}$ represents the airfoil shape definition, while the vector $\mathbf{T}^{\mathrm{T}} = [\alpha, \delta, \tau_e]$ refers to the trim parameters as the angle of attack α, the HTP deflection angle δ and the engine thrust scaling factor τ_e. Figure 7 visualizes the OpenMDAO optimization process with the shape definition via a reduced-order model, the centralized *Aeroelastics Group*, the *CFD Functional* component for the evaluation of the optimization functional $J = C_{\mathrm{drag}}$ and the evaluation of the trimming equilibrium within the explicit component *Trimming*. Within this component the aerodynamic forces $\mathbf{C} = [C_{\mathrm{fx,\,aero}},\ C_{\mathrm{fz,\,aero}},\ C_{\mathrm{my,\,aero}}]^{\mathrm{T}}$ are associated with the trim parameters α and τ_e, resulting into the optimization's equality constraint

$$\mathbf{E}_{\mathrm{trim}} \quad = \quad \begin{bmatrix} E_{\mathrm{fx}} \\ E_{\mathrm{fz}} \\ E_{\mathrm{my}} \end{bmatrix} \quad = \quad \begin{bmatrix} C_{\mathrm{fx,\,aero}}(\mathbf{D},\,\delta) \;+\; C_{\mathrm{fx,\,grav}}(\alpha) \;+\; C_{\mathrm{fx,\,engine}}(\tau_e) \\ C_{\mathrm{fz,\,aero}}(\mathbf{D},\,\delta) \;+\; C_{\mathrm{fz,\,grav}}(\alpha) \;+\; C_{\mathrm{fz,\,engine}}(\tau_e) \\ C_{\mathrm{my,\,aero}}(\mathbf{D},\,\delta) \;+\; C_{\mathrm{my,\,engine}}(\tau_e) \end{bmatrix} \quad = \quad \mathbf{0} \quad . \tag{17}$$

Static aeroelastic coupling Figure 8 expands the *Aeroelastics Group*, visualizing the static aeroelastic coupling procedure. Here it can be seen that the disciplinary components of *CFD Mesh Deformation*, *CFD Solver* and *CSM Solver* need to be implemented into implicit components, evaluating their state variables, and their corresponding

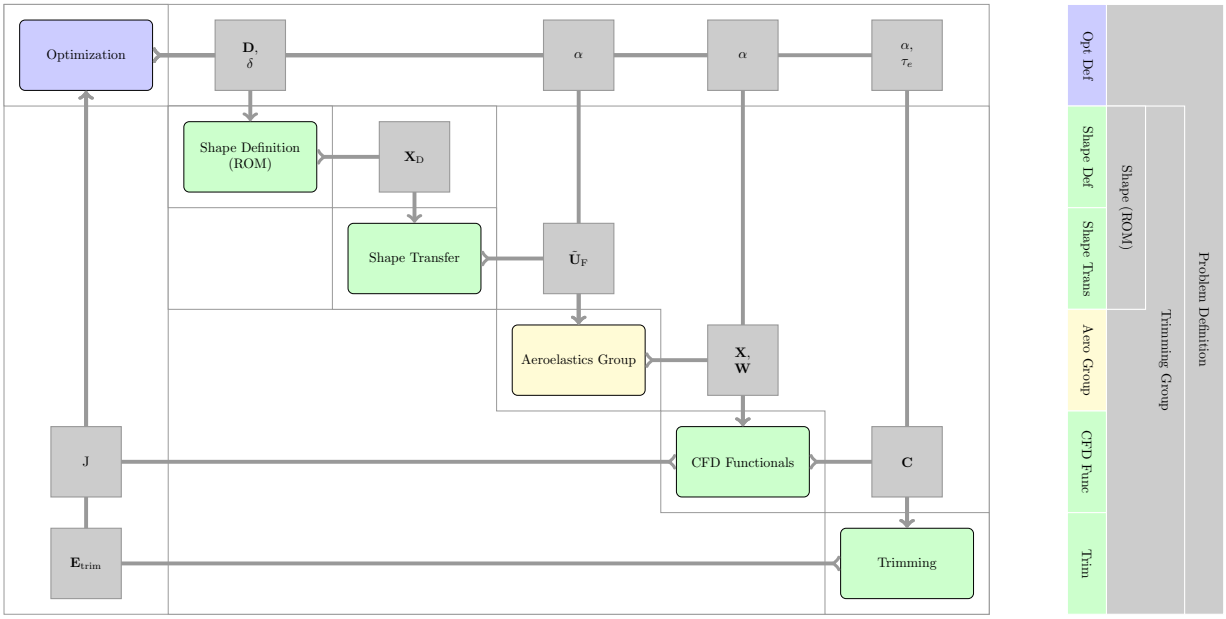

Figure 7 XDSM of complete aeroelastic MDO problem with trim constraints

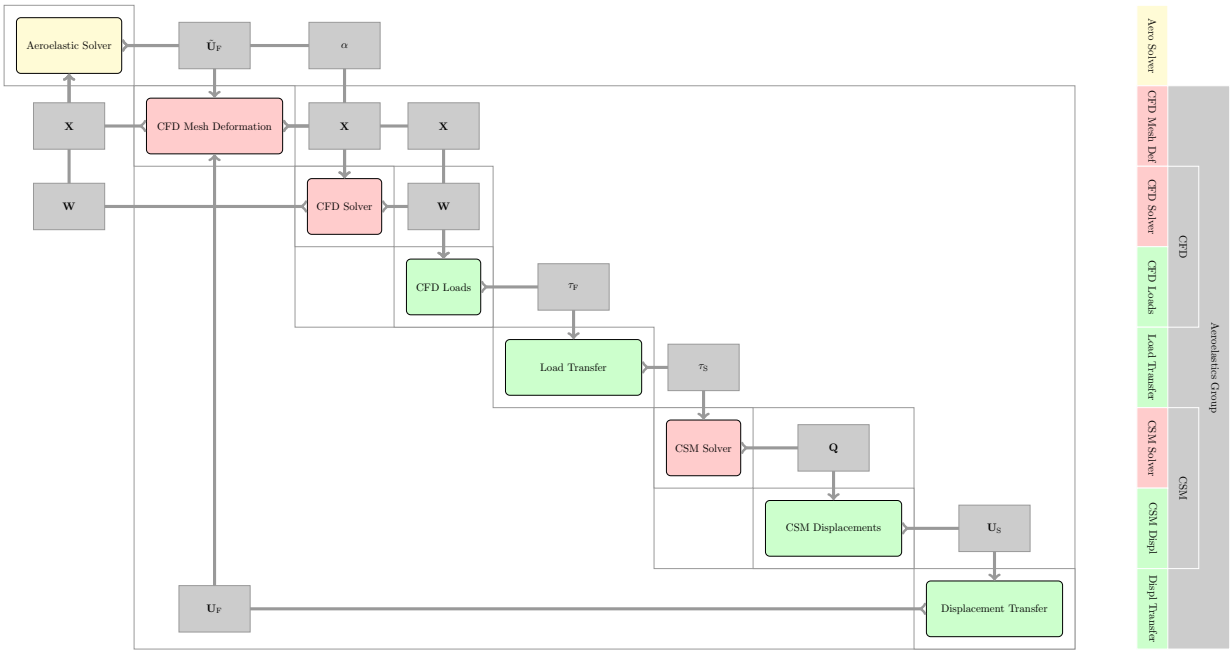

Figure 8 XDSM with expanded aeroelastic coupling group

explicit functional evaluations. A displacement vector $\tilde{\mathbf{U}}_F$ is passed from the previous shape definition into the implicit *CFD Mesh Deformation* component which morphs the initial CFD mesh towards a deformed CFD mesh $\mathbf{X}_F$. This vector is used by the CFD solver TAU within the implicit *CFD Solver* component to evaluate the CFD states vector $\mathbf{W}_F$. The explicit *CFD Loads* component then defines the forces $\boldsymbol{\tau}_F$ on the aircraft's surface nodes of the CFD mesh and hands it over to the explicit *Load Transfer* component. Here, these CFD surface mesh force vectors are interpolated onto CSM force vectors, being defined on the CSM suface mesh. The implicit *CSM solver* solver component and its corresponding explicit *CSM Displacements* component calculate the resulting mesh displacements vector $\mathbf{U}_S$, still being defined within the CSM surface mesh. Finally the explicit *Displacement Transfer* component interpolates this displacement vector into the CFD surface mesh and passes the vector $\mathbf{U}_F$ back into the implicit *CFD Mesh Deformation* component.
The coupling loop described above is embedded in the *Aeroelastics Group*. A nonlinear block Gauss–Seidel solver is added for the iterative solution of the group.

Framework-based gradient evaluation To evaluate the gradients for the objective functional J and the equality constraints $\mathbf{E}$ w.r.t. the design parameter $\mathbf{D}$ and the trim parameter $\mathbf{T}$, the adjoint method described in Section II.C is applied. In order to implement this approach into OpenMDAO the right-hand side terms of Equations (12) and (13) have to be made available to the framework. To achieve this task, the CFD method TAU is modularized in such a way that these term evaluations are accessible to the framework's corresponding component functionalities. While Figure 11 visualizes the complete linearized optimization procedure corresponding to the forward problem depicted in Figure 10, Figure 9 shows the detailed view on the components to be solved by the Equations (12) and (13).

First, the explicit *CFD Functional* component provides the terms $\frac{\partial J}{\partial \mathbf{X}}$ and $\frac{\partial J}{\partial \mathbf{W}}$ to the linear right-hand sides of the implicit *CFD Solver* and *CFD Mesh Deformation* components based on OpenMDAO's `ExplicitComponent` functionality `compute_partials()`. The explicit component *CFD Loads* uses the framework's `compute_jacvec_product()` function to provide the vector $\boldsymbol{\tau}_\mathrm{F}$ for the matrix-vector-product of

$$\left(\frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{W}}\right)^\mathrm{T} \boldsymbol{\tau}_\mathrm{F}^* \quad \text{and} \quad \left(\frac{\partial \mathbf{R}_{\boldsymbol{\tau}_\mathrm{F}}}{\partial \mathbf{X}}\right)^\mathrm{T} \boldsymbol{\tau}_\mathrm{F}^* .$$

The final missing term

$$\left(\frac{\partial \mathbf{R}_\mathbf{W}}{\partial \mathbf{X}}\right)^\mathrm{T} \mathbf{W}^*$$

of the implicit *CFD Mesh Deformation* component's right-hand side is also provided in a matrix-free way by using OpenMDAO's function `compute_jacvec_product()` for the implicit *CFD States* component, based on the frameworks input $\mathbf{W}^*$.

When the framwork API is implemented by the plug-ins or components—Equations (1 to (8)—in a fine-grained way, OpenMDAO is able to facilitate and automate the term assembly described in Equations (12) and (13). Hence, it is not necessary anymore to manually code the computation of the individual right-hand side contributions according Equations (12) and (13). This also provides the user the additional freedom to add additional disciplinary components or change the order of the existing procedure without the need of reforming the manually assembled adjoint equation system. The framework will handle this task automatically.

The ability to evaluate huge matrix-vector product in a matrix-free way is an important efficiency factor in conjunction with HPC components such as the CFD-solver TAU. When the OpenMDAO component methods `apply_linear()` and `compute_jacvec_product()` are implemented by the three implicit components and the associated explicit components, respectively, huge Jacobian matrices do not have to be allocated and assembled in the framework.

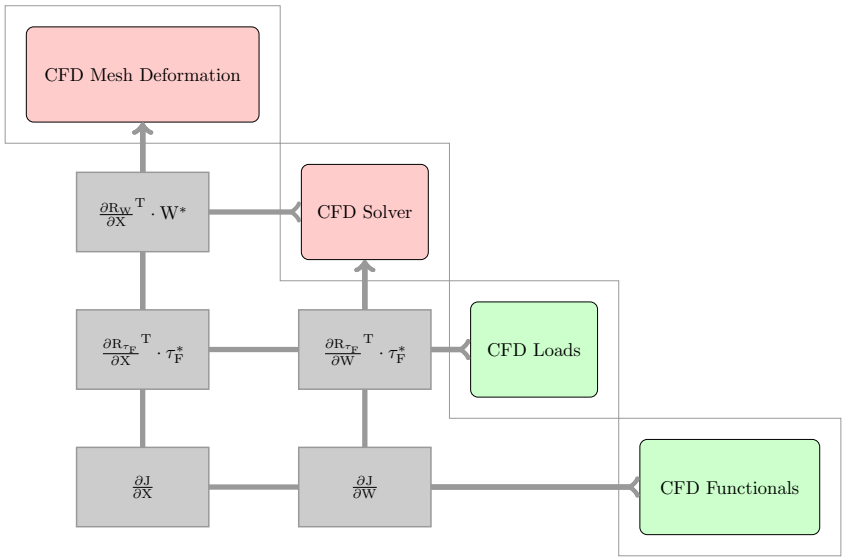

Figure 9 XDSM for adjoint problem showing the reverse-mode dependencies of *CFD Mesh Deformation* and *CFD Solver*

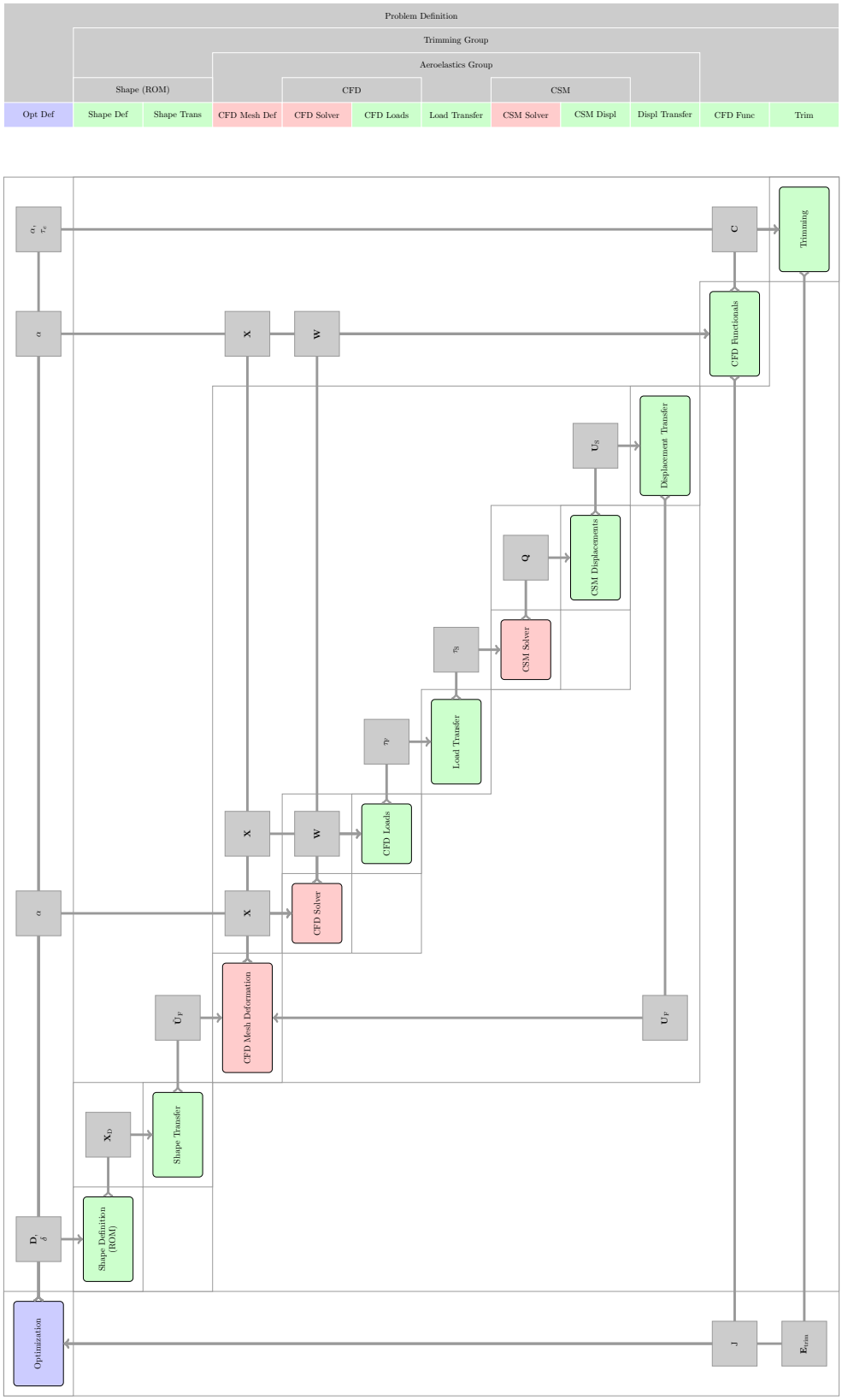

Figure 10 XDSM of overall optimization problem

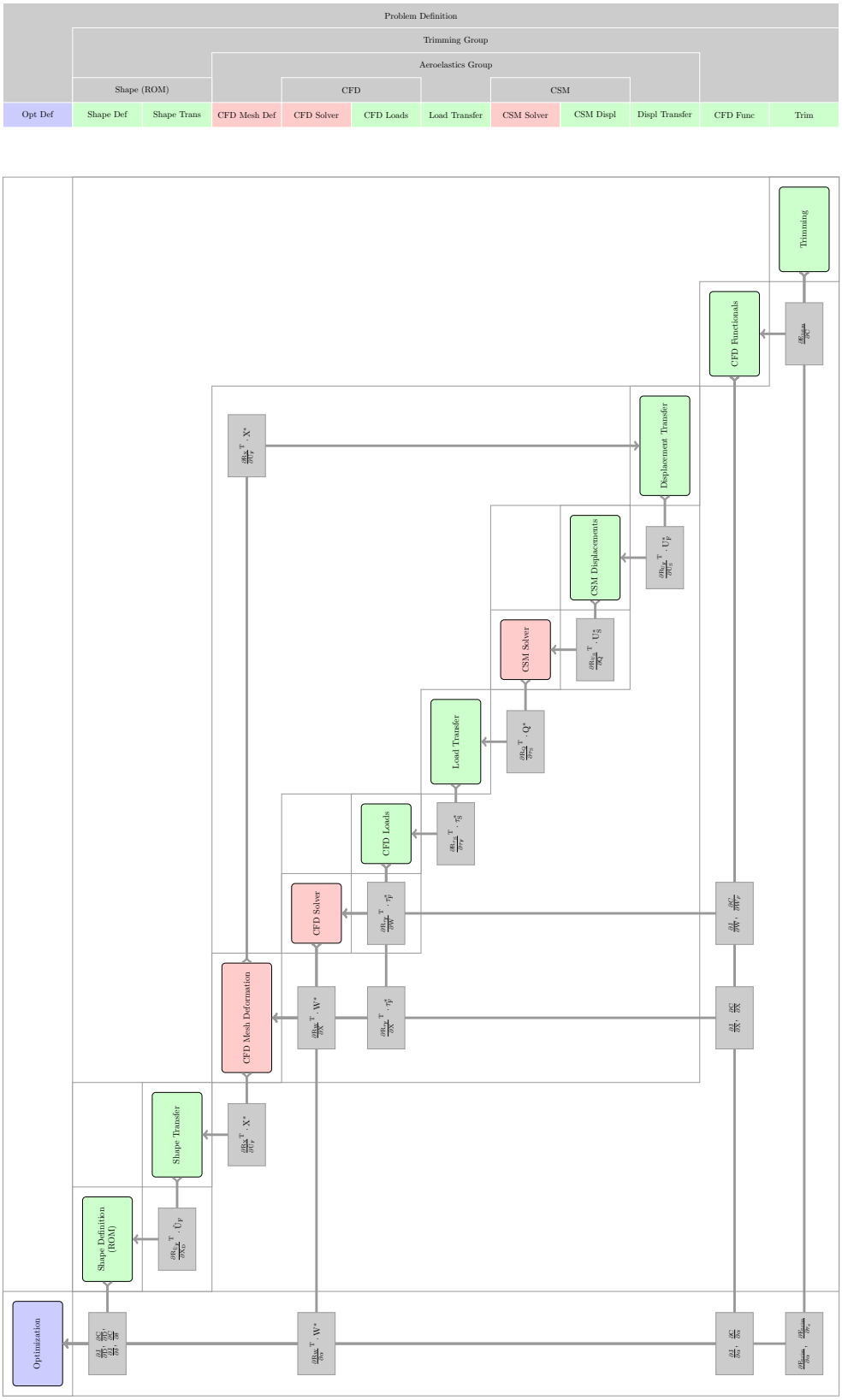

Figure 11 XDSM of overall adjoint problem

V. Evaluation of Framework Overhead for Static Aeroelastic MDO Problem

The MDO case and the optimization results previously published by Merle et al. [5] are summarized in Section V.A in order to explain the background and give the context of the high-fidelity MDO application considered for the evaluation of the suggested framework approach. In Section V.B, selected benchmark results for the `FSDMVector` implementation introduced in Section III.C are presented. This is followed by an evaluation of the potential overhead associated with the framework-based evaluation of cost functionals and sensitivity derivatives compared to the baseline FlowSimulator procedure.

A. Baseline Optimization and Results

The static aeroelastic baseline process described in Section II was presented earlier by Merle et al. [5] to minimize the drag of a generic transport aircraft configuration for trimmed cruise flight conditions. The same simulation components (FlowSimulator-plug-ins) involved were also applied in this study for the framework-based MDO approach. Details on the corresponding FlowSimulator mesh objects are presented in Table 1. Full payload and a fuel mass of 15 per cent maximum take-off weight were assumed. The engines were modeled as flow-through nacelles in conjunction with an externally defined, scalable engine thrust vector. Using the cpacs-MONA model generation process presented by [19], the loads were computed and the aircraft configuration was initially sized. A sequential quadratic programming (SQP) optimization strategy was pursued to minimize the drag. So-called trim-constrained sensitivity derivatives were computed by the static aeroelastic procedure described above and passed to the optimizer. The optimization was conducted on a sequence of meshes L1–L3 with 3.4 to 10.8 million grid nodes, cf. Table 1. To reduce the computational effort, a mesh ramping strategy was applied in the optimization starting on the coarse mesh L1. The optimization histories obtained for two different approaches are plotted in Figure 12: the red curve was obtained in conjunction with a simplified approach neglecting the aeroelastic coupling in the adjoint problem, which is referred to as "rigid adjoint" in the plot. After 31 optimization cycles the drag was reduced by 6 per cent on grid level L3. The pressure distribution over the surface of the baseline (left) and the optimized shape (right) are shown in Figure 13. After an extension of the adjoint implementation to include the complete coupling in the multidisciplinary adjoint problem, the optimization history for "Coupled Adjoint" plotted in Figure 12 was obtained. The improved gradient accuracy significantly improved the performance of the gradient-based optimization, which is shown for the grid-levels L1 and L2.

B. Framework-based Evaluation and Sensitivity Analysis

Micro benchmarks for the FSDMVector implementation In order to assess the influence of the new `FSDMVector` implementation on the optimization process runtime, we conducted a series of micro benchmarks against OpenMDAO's `PETScVector` and the new `FSDMVector`. This allows for an isolated comparison between the two vector implementations, since in the full MDO processes considered, the runtime is dominated by the simulation plug-ins. Thus, the wall-clock runtime was measured for (a) a simple L^2 norm calculation, and (b) a skeleton Arnoldi iteration (involving repeated inner products and vector projections, see Algorithm 1) for the complete top-level input vector of the OpenMDAO problem as described in Section IV. We consider the skeleton Arnoldi iteration as a typical ingredient found in more advanced multidisciplinary solution algorithms like Newton–Krylov methods to be enabled by the framework approach in the future. Note that the degrees of freedom of all disciplinary components are contained in the top-level input vector. With the considered static aeroelastic MDO problem, the size of the input vector was 64, 144, 223, 322 millions for the grid levels L0, L1, L3 and L3, respectively.[†] As shown in Figure 14 and Figure 15, both vectors show very consistent performance characteristics over all chosen mesh resolutions and core counts. For most cases it is observed that OpenMDAO performed better when it operates in place on FSDM vectors compared to its performance with the standard PETSc vector, leading to a run-time ratio FSDM over PETSc smaller than one. This outcome is very promising for future applications, particularly when a fine-grained framework integration is targeted.

[†]Mind that due to the distributed-memory concept, moderately sized components like the CSM solver that are not run in parallel lead to a memory allocation on every process; in conjunction with 384 processes and a CSM mesh of 19 thousand nodes, this leads to an overall memory allocation of 7.3 million values distributed over the processes per scalar mesh quantity. As the memory is domain-distributed, this is not considered a bottleneck here.

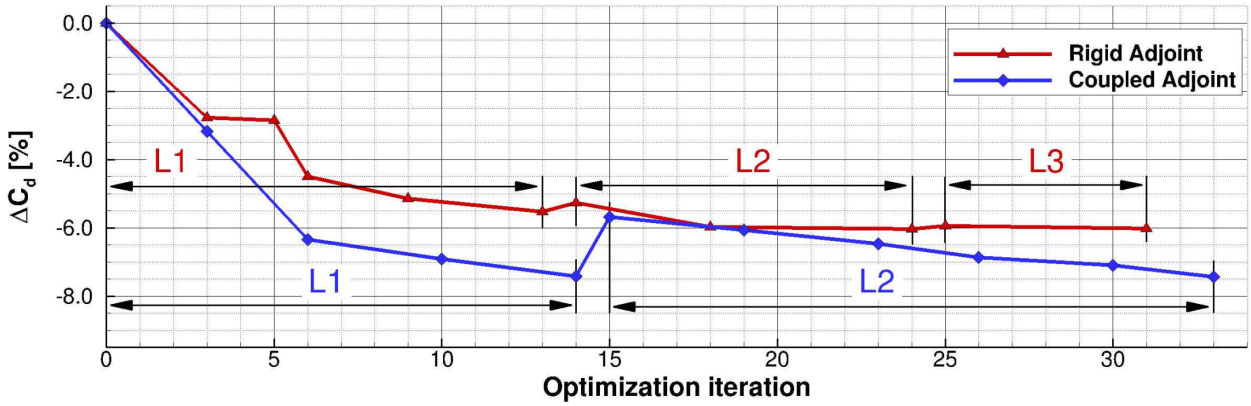

Figure 12 Reduction of the aircraft drag over the optimization cycles using a grid-ramping strategy (L1–L3). A fully-coupled aeroelastic adjoint (blue) is compared against a so-called "rigid adjoint" (red), in which the cross-coupling terms in the aeroelatic adjoint problem were neglected. The results have been shown previously [6].

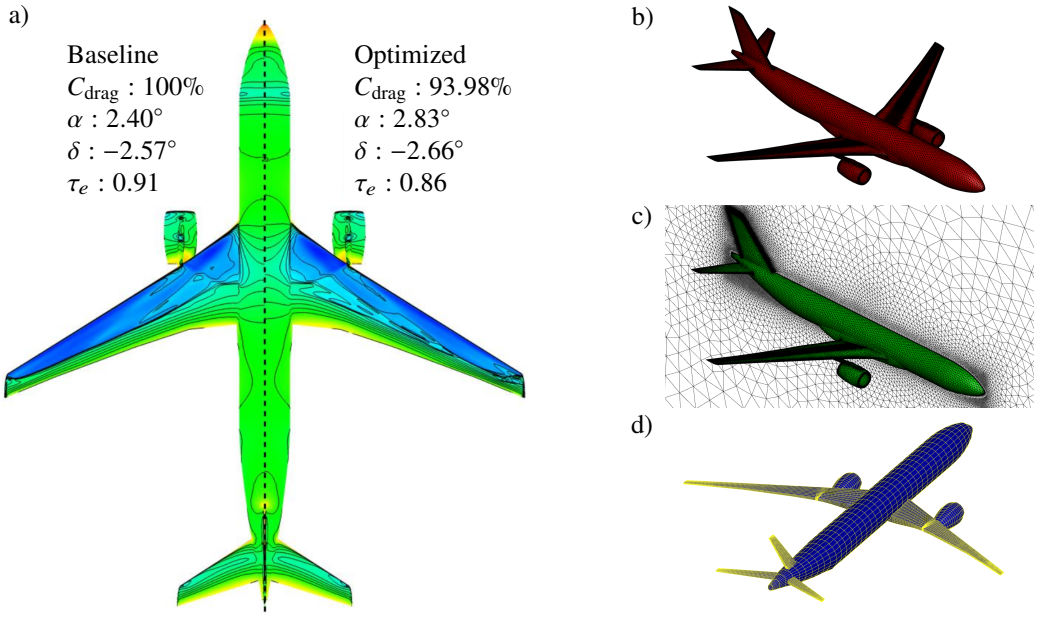

Figure 13 Initial and optimized pressure distributions obtained in a previous study [4] using the baseline MDO process (a), Shape Transfer Mesh (b), CFD Mesh (c) and CSM Mesh (d)

Table 1 Details for FlowSimulator mesh objects in the considered static aeroelastic MDO process

Index	Mesh Name		Mesh Nodes	Description
1	ROM Mesh		29 k	Spatial point distribution on the parameterized surfaces for prediction of displacements and geometric sensitivities by the parametric CAD-ROM
2	Shape Transfer Mesh	L0	21 k	Unstructured surface mesh extracted from the
		L1	86 k	*CFD Volume Mesh* to interpolate ROM predictions onto
		L2	180 k	
		L3	288 k	
3	CFD Volume Mesh	L0	544 k	CFD volume mesh with various datasets
		L1	3,385 k	e. g. coordinates, displacements, force vectors,
		L2	6,613 k	adjoint vectors, ...
		L3	10,837 k	
4	CFD Coupling Mesh	L0	41 k	Unstructured surface mesh extracted from the
		L1	171 k	*CFD Volume Mesh* to interpolate CSM
		L2	360 k	primal/adjoint displacements onto
		L3	575 k	
5	CSM Coupling Mesh		19 k	Structured surface mesh extracted from the *CSM Mesh* to interpolate CSM primal/adjoint forces onto
6	CSM Mesh		19 k	Structured CSM Mesh to calculate displacements out of given forces in primal/adjoint mode

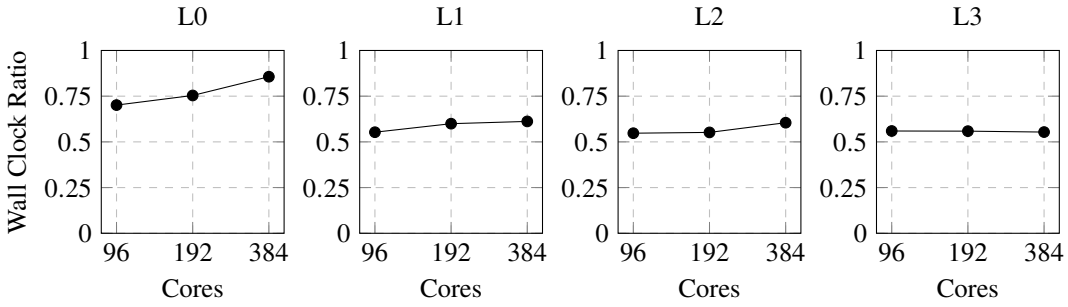

Figure 14 Ratio of wall clock times $t_{\mathrm{FSDM}}/t_{\mathrm{PETSc}}$ of L^2 norm calculation for the system input vector with the FSDM vs. the PETSc vector implementation at different mesh resolutions yielding an average improvement by factor 0.62.

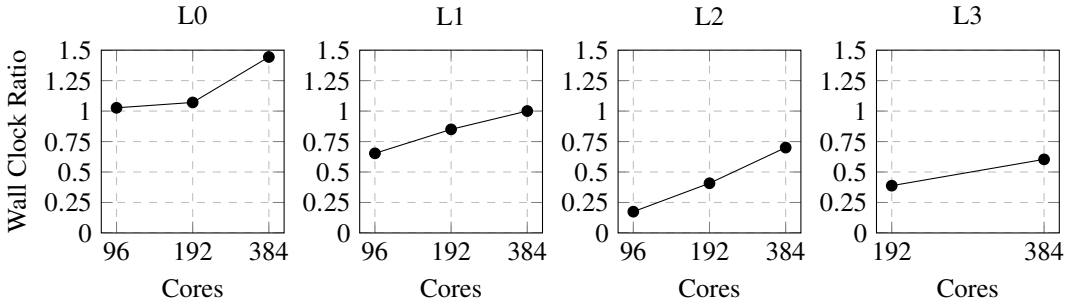

Figure 15 Ratio of wall clock times $t_{\mathrm{FSDM}}/t_{\mathrm{PETSc}}$ of Arnoldi skeleton iteration for the system input vector with the FSDM vs. the PETSc vector implementation at different mesh resolutions yielding an average improvement by factor 0.76. The L3 runs with 96 cores are omitted since the benchmark data did not fit into memory at this domain decomposition.

Algorithm 1 Skeleton Arnoldi iteration to benchmark `FSDMVector`. $\mathbf{V}$ is the model's input vector. The $\mathbf{Q}$ vectors are randomly chosen.

1: **for** $k \leftarrow 1$ to 50 **do**
2: # ...
3: **for** $j \leftarrow 1$ to k **do**
4: $h \leftarrow \langle \mathbf{V}, \mathbf{Q}_j \rangle$ ▷ using the `Vector.dot` method
5: $\mathbf{V} \leftarrow \mathbf{V} + h\mathbf{Q}_j$ ▷ using the `Vector.add_scal_vec` method
6: **end for**
7: **end for**

Runtime Comparison A run time comparison study was conducted to evaluate the overhead of the framework-based evaluation of functionals and derivatives compared to the baseline procedure. We considered one functional evaluation, Figure 10, and an adjoint-mode derivative evaluation, Figure 11. For the timings, the maximal iteration number for both the nonlinear block Gauss–Seidel solver (NLBGS) and the linear block Gauss–Seidel solver (LNBGS) was set to 5 for the FlowSimulator-standalone and FlowSimulator-OpenMDAO process. 1000 FAS multigrid iterations were carried out in the nonlinear TAU run and 300 multigrid-preconditioned Krylov sweeps in the adjoint TAU run. The adjoint run-time included four separate adjoint solutions for the evaluation of the functional $J = C_{\mathrm{drag}}$ and the three entries in the constraint vector $\mathbf{E}_{\mathrm{trim}} = [E_{\mathrm{fx}},\ E_{\mathrm{fz}},\ E_{\mathrm{my}}]^{\mathrm{T}}$.

Figure 16 shows a comparison of the overall wall clock time of one optimization iteration between the FlowSimulator-standalone and the FlowSimulator-OpenMDAO approach, executed on 288 cores using the L3 mesh, cf. Table 1. The

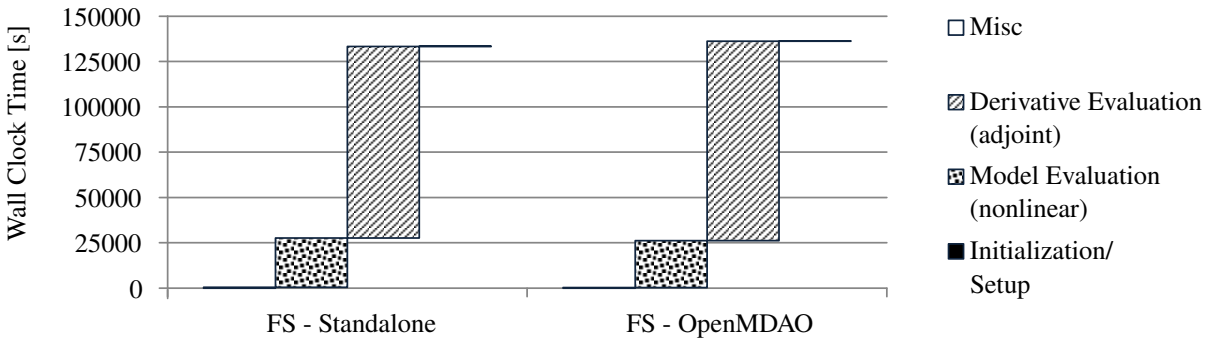

Figure 16 Run-time comparison between baseline (FS-Standalone) and framework-based (FS-OpenMDAO) process for model and derivative evaluations using the L3 mesh

run-time of the adjoint-based derivative evaluation exceeds the nonlinear evaluation of the model since four adjoint problems are solved in a row for the gradient evaluation of C_{drag}, E_{fx}, E_{fz} and E_{my}. It is important to note, that the measured run times are very similar for the FlowSimulator-standalone and the FlowSimulator-OpenMDAO approach. The observed differences are attributed to minor changes in the process definitions due to the component modularization. Thus, Figure 16 indicates that—even for the fine-grained and modular framework integration suggested here—the overhead caused by the use of the OpenMDAO framework in the evaluations of functionals and derivatives in adjoint mode is very small compared to the FlowSimulator standalone implementation. We made a very similar observation looking at the run-time breakdown into disciplinary contributions, cf. Figure 17: runnig FlowSimulator together with OpenMDAO framework did not lead to significant relative changes compared to the FlowSimulator standalone implementation.

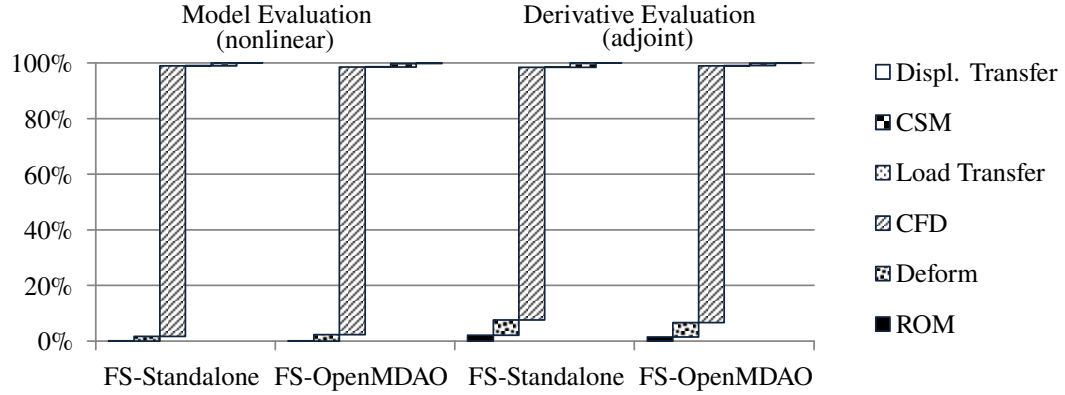

Figure 17 Break-down of run times for individual disciplinary components in combination with baseline (FS-Standalone) and framework-based (FS-OpenMDAO) processes on L3 mesh

VI. Conclusions

A framework-based MDO software architecture was suggested and evaluated in this paper that integrates the FlowSimulator HPC environment and the MDO framework OpenMDAO in a seamless manner. The FSDM as a parallel in-memory data management centrally shared by the entire ecosystem FlowSimulator is carried over to the integrated FlowSimulator-OpenMDAO approach. With a library extension presented in this paper, OpenMDAO is enabled to directly operate on central FSDM mesh objects in fully parallel workflows. Working in-place on FSDM data avoids data redundancies and elementary mesh functionalities centrally provided by the FSDM for the shared data can be reused by the MDO ecosystem. In dedicated mircro benchmarks, we observed a very good performance of the OpenMDAO framework operating in place on FSDM mesh objects. The framework method was evaluated based on an existing high-fidelity MDO procedure for an aeroelastic transport aircraft configruation in trimmed cruise conditions. The individual FlowSimulator plug-ins were integrated in a modular way into the suggested MDO framework consisting of OpenMDAO and FlowSimulator. Since the methods of the FlowSimulator ecosystem were exposed to the framework in a fine-grained way, the computation of sensitivity derivatives for multidisciplinary problems in adjoint mode could be significantly facilitated using the MAUD strategy implemented in OpenMDAO. Our first evaluation studies using the high-fidelity MDO scenario indicate that the overhead associated with the suggested framework approach FlowSimulator-OpenMDAO is negligible compared to the baseline FlowSimulator implementation of the MDO process, in which the multidisciplinary gradient computations in adjoint mode were hand-coded.

In the next step, the framework approach evaluated here for the adjoint-based gradient computation will be applied to drive the full optimization. Moreover, it is intended to include structural sizing in the framework-based MDO. Another topic for future work enabled by the framework-based approach is the use of more advanced, hierarchical multidisciplinary solution algorithms for high-fidelity MDAO problems.

References

[1] Gray, J. S., Hwang, J. T., Martins, J. R., Moore, K. T., and Naylor, B. A., "OpenMDAO: An open-source framework for multidisciplinary design, analysis, and optimization," *Structural and Multidisciplinary Optimization*, Vol. 59, No. 4, 2019, pp. 1075–1104.

[2] Reimer, L., "The FlowSimulator – A Software Framework for CFD-related Multidisciplinary Simulations," *European NAFEMS Conference: CFD – Beyond the Solve*, Munich, Germany, 2015.

[3] Hwang, J., and Martins, J., "A computational architecture for coupling heterogeneous numerical models and computing coupled derivatives," *ACM TOMS*, Vol. 44, New York, NY, USA, 2018.

[4] Merle, A., Ronzheimer, A., Bekemeyer, P., Görtz, S., Keye, S., and Reimer, L., "Gradient-based Optimization of a flexible long-range transportation aircraft using a high-dimensional CAD-ROM parametrization," *Proceedings of Deutscher Luft- and Raumfahrtkongress (DLRK)*, Friedrichshaven, Germany, 2018.

[5] Merle, A., Ilic, C., Abu-Zurayk, M., Jäßy, J., Becker, R., Schulze, M., and Klimmek, T., "High-Fidelity Adjoint-based Aircraft Shape Optimization with Aeroelastic Trimming and Engine Coupling," *EUROGEN Conference*, Guimaraes, Portugal, 2019.

[6] Backhaus, T., Gottfried, S., Ilic, C., Merle, A., and Stück, A., "Integration of High-Fidelity Analysis Tools in MDO Frameworks for HPC," *AIAA 2019-3107*, Dallas, TX, USA, 2019.

[7] Merle, A., Stueck, A., and Rempke, A., "An Adjoint-based Aerodynamic Shape Optimization Strategy for Trimmed Aircraft with Active Engines," *AIAA 2017-3754*, Denver, CO, USA, 2017.

[8] Bobrowski, K., Ferrer, E., Valero, E., and Barnewitz, H., "Aerodynamic Shape Optimization using Geometry Surrogates and Adjoint Method." *AIAA Journal*, Vol. 55, No. 10, 2017, pp. 3304–3317.

[9] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H., "PETSc Web page," `https://www.mcs.anl.gov/petsc`, 2019.

[10] Kroll, N., Langer, S., and Schwöppe, A., "The DLR Flow Solver TAU – Status and Recent Algorithmic Developments," *AIAA 2014-0080*, National Harbor, MD, USA, 2014.

[11] Stück, A., "An adjoint view on flux consistency and strong wall boundary conditions to the Navier-Stokes equations," *J. Comput. Phys.*, Vol. 301, 2015, pp. 247–264.

[12] Stück, A., "Dual-consistency study for Green-Gauss gradient schemes in an unstructured Navier-Stokes method," *J. Comput. Phys.*, Vol. 350, 2017, pp. 530–549.

[13] Klimmek, T., Schulze, M., Abu-Zurayk, M., Ilic, C., and Merle, A., "cpacs-MONA - An Independent and in High-Fidelity Based MDO Tasks Integrated Process for the Structural and Aeroelastic Design of Aircraft Configurations." IFASD Conference, Savannah, GA, USA, 2019.

[14] Lancaster, P., and Salkauskas, K., "Surfaces generated by Moving Least Squares Methods," *Math. Comput.*, Vol. 37, 1981, pp. 141–158.

[15] Mader, C. A., Kenway, G. K., Yildirim, A., and Martins, J. R., "ADflow: An Open-Source Computational Fluid Dynamics Solver for Aerodynamic and Multidisciplinary Optimization," *Journal of Aerospace Information Systems*, Vol. 17, No. 9, 2020, pp. 508–527.

[16] Roy, S., Moore, K., Hwang, J. T., Gray, J. S., Crossley, W. A., and Martins, J., "A mixed integer efficient global optimization algorithm for the simultaneous aircraft allocation-mission-design problem," *58th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference*, 2017, p. 1305.

[17] Hwang, J. T., Jasa, J. P., and Martins, J. R., "High-fidelity design-allocation optimization of a commercial aircraft maximizing airline profit," *Journal of Aircraft*, Vol. 56, No. 3, 2019, pp. 1164–1178.

[18] Hwang, J., "A modular approach to large-scale design optimization of aerospace systems," *Ph.D. thesis*, University of Michigan, USA, 2015.

[19] Klimmek, T., Bach, T., Bramsiepe, K., Dähne, S., Jepsen, J., Kier, T., Kohlgrüber, D., Leitner, M., Petsch, M., Schulze, M., and Schuster, A., "Overall Aircraft Design Synthesis, Loads Analysis, and Structural Optimization – Parameterized Disciplinary Sub-Processes within High-Fidelity-based Aircraft MDO," , 4–6 September 2018.