ACADÉMIE
DES SCIENCES
INSTITUT DE FRANCE

Comptes Rendus

Mécanique

Michael Lienhardt and Bertrand Michel

SoNICS: design and workflow of a new CFD software

Volume 353 (2025), p. 1261-1287

Online since: 1 December 2025

https://doi.org/10.5802/crmeca.334

CENTRE
MERSENNE

The Comptes Rendus. Mécanique *are a member of the*
Mersenne Center for open scientific publishing
www.centre-mersenne.org — e-ISSN : 1873-7234

ACADÉMIE
DES SCIENCES
INSTITUT DE FRANCE

Comptes Rendus. Mécanique
2025, Vol. 353, p. 1261-1287
https://doi.org/10.5802/crmeca.334

Research article

SoNICS: design and workflow of a new CFD software

Michael Lienhardt [*], [a] **and Bertrand Michel** [b]

[a] DTIS, ONERA, Université Paris-Saclay, 91120 Palaiseau, France
[b] DAAA, ONERA, 92320 Châtillon, France
E-mails: michael.lienhardt@onera.fr, bertrand.michel@onera.fr

Abstract. SoNICS is a new CFD software which builds upon the experience acquired in the years developing and extending the *elsA* CFD tool implemented at ONERA and used in industry. This article presents an overview of the SoNICS design and workflow, the motivation for this new tool, its structure and how it manages the user configuration (including the input mesh) to produce in the end a graph of tasks to schedule. In particular, we will discuss how well-studied concepts like variability, task graphs and compilations passes are cornerstones of the SoNICS workflow and allow for: a relatively easy evolution and maintenance of the code-base; a relatively straightforward user experience; and good performances.

Keywords. Computational fluid dynamics, variability, code assembly, compilation passes.

Note. Article submitted by invitation.

Manuscript received 19 January 2025, revised 9 October 2025, accepted 20 October 2025.

1. Introduction

elsA is an important Computational Fluid Dynamics (CFD) solver that was improved and extended over more than 25 years of development [1–3]. It was and still is developed in partnership with and used by several industrial partners such as Airbus, Safran, EDF or MBDA. *elsA* is now a tool with many capabilities, and can be used both for transport aircraft configurations [4,5] and helicopter applications [6,7], as well as for turbomachinery flow simulations [8,9] and steam turbine applications [10,11].

However, *elsA*'s implementation was not designed to tackle the challenges that were raised in the past decade. Indeed, it has a monolithic structure based on oriented object approach that combines python, C++ and Fortran code with intricate dependencies. This structure together with the large code base implementing the many functionalities supported by *elsA* makes it difficult to maintain and extend, which includes adding support for new efficient hardwares such as GPUs. Moreover, to support gradient optimization, the code differentiation of all kernels in *elsA* must be hand-written instead of automatically generated using either existing static tools [12,13] or dynamic analysis [14–16]. Updating *elsA* to support GPUs and automatic differentiation would require a complete overhaul of its implementation.

[*] Corresponding author

In addition to these difficulties, the usecases for fluid flow simulation evolved in the past decades, and new functionalities are expected by our industrial partners or by internal requests from ONERA's teams. In particular, we identified three main requests from *elsA*'s users.

Simulations: There is an important interest in simulating the flow around or inside larger objects and with more precision, including the simulation of full engines. Such simulations are complex to configure and require much memory and computational resources to perform. A modern CFD software must be easy to configure and must be able to take advantage of powerful hardware, that are all complex, distributed and heterogeneous.

User experience: Setting up a simulation, checking that it is going smoothly and analysing the results are arduous tasks, and a CFD tool should integrate tools and techniques that would make them as straightforward as possible. This includes simple mechanisms to help the user to design their mesh (such as automatic mesh refinement), and APIs to help couple the tool with other solvers. Many elements can contribute to a better user experience (e.g., better user interfaces, clear and interactive feedbacks), but all require a robust understanding of what is the information relevant to the user, what computation is being performed, and how to transform this computation to perform additional task, like exchanging data with another solver or computing mesh refinement metrics and updating the mesh.

Design: Simulations can be used to help design objects by pointing what features of a simulated artifact cause turbulences or other unwanted behaviors. This is done by differentiating the performed simulation, and so a CFD tool should provide a general mechanism to produce such derivatives so the end users can get all the data they need to analyse and optimize their artifact.

This paper presents the new CFD tool SoNICS which builds upon the expertise developed around *elsA*, and uses a new implementation design based on technics inspired from software engineering, scheduling, and other to solve the limitations of *elsA*'s implementation and answer our partners' requests.

This article is structured as follows: Section 2 presents the main choices made in the design of SoNICS and motivates how they contribute in satisfying the user requests; Section 3 briefly presents and discusses some of the simulation that were already done using SoNICS; Section 4 discusses the related work and Section 5 concludes the article.

2. SoNICS's design and workflow

As shown by the work around *elsA* and other CFD solvers, designing such a tool is a very difficult task. The main difficulty is clearly captured by the apparent contradiction raised by the **Simulations** requirement given in Section 1. Indeed, a CFD solver answering this request must be able to perform a large panel of computations on different hardware architectures, which requires to have a large configuration space (e.g., which turbulence model to apply, how many CPUs are available), and a complex control flow to apply this configuration during the computation. This control flow entails a large running overhead, which contradicts the requirement for the computation to be efficient. Moreover, compilers also have difficulties to manage codes with complex control flow, e.g., even if the field of code vectorization is the subject of much research [17–19], all the technics must have a complete knowledge of the loops to vectorize and fail with conditionals. Finally, automatic differentiation, required by the **Design** requirement, can be achieved on code with complex control flow with a well known technic [14–16] which unfortunately relies on registering at runtime all performed operations, which can create a large memory overhead as discussed in [20]. On the opposite, purely functional code can easily be differentiated using source transformation [12,13] with no overhead during execution.

A second difficulty is captured by another contradiction raised by the **Simulations** and **User experience** requirements given in Section 1. Indeed, on one hand, the **Simulations** requirement asks a CFD solver to be able to perform a large range of simulations, which implies providing to the user an extensive panel of ways to configure the solver, including many options and hooks into its code. On the other hand, the **User experience** requirement asks a CFD solver to provide every user with a clear and expressive interface dedicated to their needs, to help them as much as possible in setting up their simulation and pinpointing configuration errors before starting any computation.

The development of *elsA* faced these difficulties at a lesser level, and proposed an interesting solution for the main difficulty, that extracts its control flow from its computation. Indeed, the control flow of a CFD solver has three main objectives: (i) it selects which variant of a functionality to execute, depending on the user configuration (e.g., which turbulence model to apply); (ii) it distributes the computation over the hardware architecture and identifies the necessary communications between the nodes; and (iii) it triggers the computation of values required by a functionality, depending on which variant of the functionality is being executed and on the mesh topology. Since none of the inputs of the control flow change during the computation, the selection, distribution and triggers can be computed once and for all, and the computation can then be performed without any overhead. *elsA* implemented its control flow in a complex component called *Factory*, and the computation's implementation was split into many independent kernels performing a well-identified task (e.g., computing the gradient of a value), called *operators*, that the *Factory* would combine to produce a computation. This design has three additional advantages:

(1) operators are simpler to understand and implement than complete functionalities, which improves their capability to be checked, tested and maintained;
(2) many functionalities share some computations, and so splitting them into operators performing these computations avoids code duplication and also avoids possibly computing multiple time the same data at runtime;
(3) it is easier to update, extend and add new functionalities with such a modular implementation, since existing operators can be used in new functionalities, and new ones can be seamlessly integrated.

However, *elsA*'s *Factory* is difficult to maintain and extend, has difficulties to manage complex distribution schemes, and is completely unable to handle automatic differentiation.

SoNICS's design uses the same approach to keep *elsA*'s interesting properties, but completely replaces the *Factory* with an explicit workflow based on the three following technics:

(1) it uses a standard and well studied formalism to manage its many configuration options called *Software Product Lines* (SPLs) [21];
(2) it models the computation as an acyclic graph of operators, which enables SoNICS to perform a very large range of analysis and optimization that were not possible in *elsA*, including having a very efficient distribution and communication scheme;
(3) in particular, SoNICS currently implements four graph transformation algorithms that manage: (i) memory optimization and allocation; (ii) building the communication between the different hardware nodes; and (iii) performing the automatic differentiation.

The rest of the section is structured as follows: we first discuss how we structure SoNICS's user interface to answer the **User experience** requirement; we present the complete SoNICS workflow that replaces *elsA*'s *Factory*; and then we discuss in more details the different aspects of the acyclic graph of operators and its transformations.

2.1. *User interface*

Configuring a simulation is a difficult task, which involves setting up which physical model to use, which numerical scheme, which constants to use, and creating a mesh that could capture the phenomena of interest. Unfortunately, no solution was found to simplify the configuration process for a general purpose CFD tool. However, similar simulations are usually also configured similarly, and so it makes perfect sense to design a wrapper tool specific for a category of simulations that would only ask the user for the few information that would change in configuring a simulation of that category, and automatically extend it into a full configuration for the CFD tool.

To simplify the direct configuration of SoNICS and the design of wrapper tools, we identified from our experience with *elsA* the central elements in configuring a simulation, and designed a way to make these elements as natural to use as possible, and also focused on error reports, so the user, if their configuration is erroneous, could easily identify what the problem is and how to solve it.

SoNICS's user interface is constructed around four configuration elements.

Topological data: We use the CGNS (CFD General Notation System) standard [22,23] to store the topology and its associated data. This format is widely used for storing and sharing CFD data (mesh structure in particular) and provides a clear hierarchical organization of data, facilitating the management of complex mesh structures with a clear separation of geometry, topology and solution data. Moreover, ONERA has developed the *Maia* library [24] which facilitates the manipulation of CGNS data structures using a simple python API. *Maia* provides easy access to complex algorithms such as load/store operations, partitioning, and solution transferring within a distributed memory context. Finally, to enhance modularity and performance in distributed memory environments, ONERA has implemented a simple and efficient in-memory representation of the CGNS format. This approach allows for efficient data handling and manipulation, which is crucial for high-performance computing applications. The integration of the in-memory CGNS representation is effectively utilized in the SoNICS framework, supporting the entire workflow from pre-processing to post-processing.

Options: As discussed previously, we use SPL technics to manage the organise the many options of SoNICS, and in particular, these options are organised into a *Feature Model*. Feature Models have the interesting property of capturing the notion of *valid* option selections. Indeed, not all options can be selected together. This property allows SoNICS to detect and report possible errors in the user's option selection before starting its computation.

Data of interest: The user may be interested in data that are not part of the computation of the fluid flow, e.g., the pressure on a specific boundary to check if the material is resistant enough, or some metrics used for automatic mesh refinement. *elsA* uses an enumeration to list all data that can be requested by the user, which forbids the user to request data that could be computed but that was not included in the enumeration. In SoNICS, we designed a *Domain Specific Language* (DSL) embedded in python to express any possible data: this way, any data that can be computed by SoNICS can be requested by the user.

Workflow modification: Finally, some user requests have an impact on SoNICS's workflow. First, coupling with another solver requires to regularly exchange data with another software: to this end, SoNICS includes a very simple *Aspect Oriented Programming* [25–27] interface called *triggers* which allows the user to insert any python code before and after each iteration of the simulation. Moreover, automatic mesh refinement wraps the whole SoNICS's workflow in a loop where each iteration computes some refinement metrics and adapts the mesh w.r.t. these metrics, or exits the loop when the obtained mesh is satisfactory. To allow such an intrusion into SoNICS's workflow, it has been implemented in python with clear modules and data structures, so any motivated user could insert their code where it is relevant.

2.2. *Complete workflow*

The complete workflow of SoNICS is presented in Figure 1. It produces from the user configuration a well-balanced distributed computation performing the requested simulation. The workflow starts from the left: we have the *Operator Bank*, filled by the SoNICS developers, which stores all the information concerning the operators available in SoNICS; and the user configuration split into the *Option Selection*, the *CGNS input file* and the *Other user requests*. The *Option Selection* is used by the *Variant Selection* mechanism to obtain the *Operator Variants* from the *Operator Bank*. This mechanism is a standard tool in the SPL formalism, which will be discussed in Section 2.3. In parallel to the *Variant Selection*, the raw mesh information, called *DistTree* is extracted from the CGNS file. Using the *DistTree*, the *Variant Selection* and the data of interest in the *Other user requests*, we generate a *Primal Operator Graph* whose purpose is to identify the computation that needs to be performed, and create a uniform distribution of that computation. This distribution is computed by the *Distribution* component, which is currently implemented using SCOTCH [28,29] or ParMETIS [30,31], and is used to produce a distributed mesh called *PartTree*. Using this distributed mesh, SoNICS produces a *Distributed Operator Graph*: this is the graph that will be used during SoNICS's computation. This distributed graph is then managed by the *Builder* which fetches the implementation of the different operators, gets from the *PartTree* the different constants needed by the operators, resolves the inputs and outputs of the operators into actual pointers, and produces a distributed *taskflow* that can be executed by the runtime. The runtime is thus responsible of performing the computation given in the input taskflow, and also performs the additional tasks requested by the user in the *Other user requests*. It is composed by many libraries: some, like *ParaDiGM* [32], are used to manage the mesh, the data hosted by it and the communications between the different computation nodes; some, like *taskflow* [33] are used to execute the graph both on CPU and on GPU; others, like the *array manager* are internal to SoNICS to perform various tasks (e.g., the *array manager* implements all of the data allocation and reallocation functionalities required to execute the graph's operators).

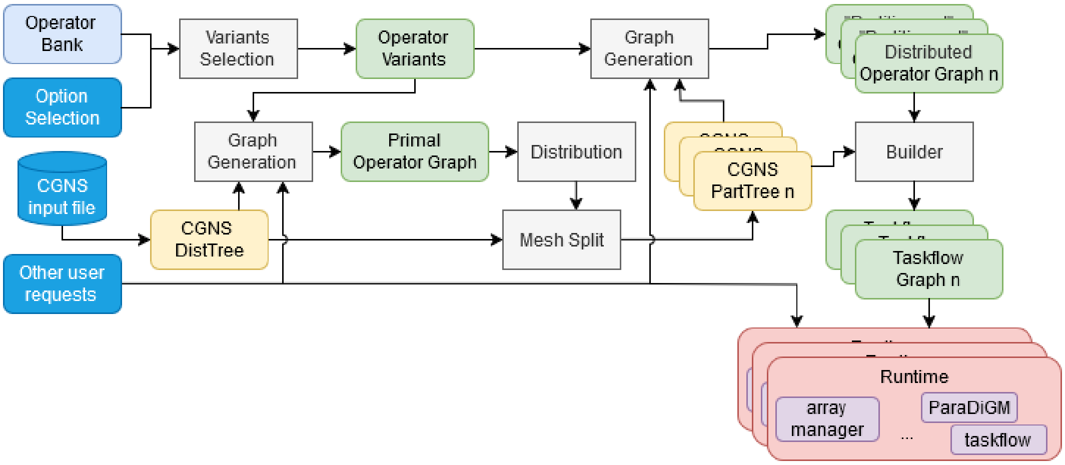


Figure 1. SoNICS's main workflow.

2.3. *Software product lines*

Conceptually, a Software Product Line (SPL) corresponds to a set of similar programs, called variants, that share many characteristics and only differ due to well identified feature selection [21]. For instance, the linux kernel is a well-known SPL [34]: while every running linux kernels do implement an operating system, depending on the selected and unselected options before compiling it, many of the functionalities of that system do change (e.g., hardware support, dynamic module loading, etc). SPLs are largely used in the automotive industry [35] to manage the software of the many car variant. The field of SPL focuses on solving two problems: (i) how to organize the many options of a software to have a clear picture of the configuration space; and (ii) how to apply these options in the code to avoid code duplication and facilitate maintenance and evolution.

A CFD solver such as SoNICS is a natural usecase for SPL, since it must include many options to capture all the possible user configurations. In SoNICS, we used a well-known standard to organize its options, called *Feature Model* (FM). Such a model organizes options in a tree hierarchy, with additional constraints on option selection written with boolean formula. A simplified version of SoNICS's FM is presented in Figure 2. Here, the configuration space of SoNICS (modeled with the `sonics` option) is split in four categories: `formulation` states which discretization of the computation to use (e.g., either computes the values in the cells of the mesh, or on its vertices); `spacial_dimension` states how many dimensions to consider (2 and 3 dimensions are currently supported); `motion` states if the computation is stationary or not; and `model` presents the options to configure the equations to consider in the computation. Below the tree hierarchy are two additional constraints restricting how the options can be selected: `transition_closure ⇒ kl ∨ kw` means that if the user wants a transition closure, then only the turbent model `kl` or `kw` can be selected; and `zdes ⇒ spalart` means that the Zonal Detached Eddy Simulation can only be performed in SoNICS with the spalart turbulence closure.

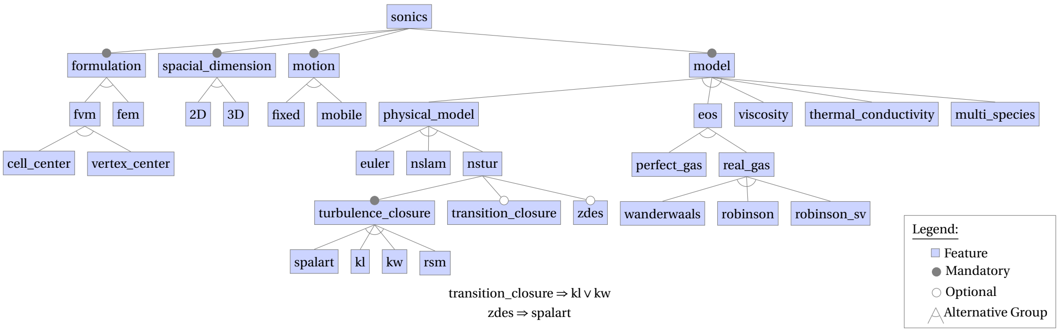


Figure 2. A simplified version of SoNICS's feature model.

The effect of most of these options is to select which variant of every operators implemented in SoNICS will be used in the simulation, and to generate some additional information that will be discussed later. This selection, performed by the *Variant Selection* in Figure 1, is implemented by a combination of *Delta-Oriented Programming* [36,37] which carries the options selection to the operators, and either brute force implementation of an operator (one implementation per variant), or code generation technics based on C++ templates or sympy [38,39].

2.4. *Task graph*

While the SPL mechanism manages the selection of the operators' variants, the operator graph captures which operators are necessary for a given computation. This graph structures the computation into two kinds of nodes, operator nodes and data nodes, and where the edges connect operators to their output data and data to the operators using them in input.

The *Graph Generation* task in SoNICS's workflow automatically produces this graph, and works by: first producing a *local operator graph* describing only the computation of the direct data; and then applying the different algorithms that will be discussed in Section 2.5 to complete the graph with derivative computation, memory allocations, etc. In this section, we present how the initial local operator graph is generated.

The first step of the *Graph Generation* is based on two important characteristics of many HPC computations: (i) the computation calculates well identified values (e.g., in SoNICS, these values are the flow update and other values explicitly requested by the user); (ii) the steps that perform this computation are mostly side-effect-free, i.e., they have well identified inputs and outputs and do not have complex interaction patterns using shared objects or global data. Using these two characteristics, the local operator graph is generated, using the operators stored in the *Operator Variants* registry with the following pseudo-algorithm:

```
todo = set of all values to be computed
done = $\emptyset$
while todo $\neq \emptyset$:
  v = remove a value from todo
  f = find an operator that computes v
  done = done $\cup$ {v}
  todo = todo $\cup$ (inputs(f) \ done)
```

This algorithm starts with the values that need to be computed (i.e., flow update and the user's data of interests) and iteratively adds operators to compute these values, these operators having themselves inputs that need to be computed. Hence, it is essential for this algorithm to have a clear description of the inputs and outputs of every available operators: this description is one of the additional information generated during the *Variant Selection* and is expressed using the DSL also used by the user to specify their data of interest. Moreover, an important property of a CFD simulation is that its computation depends on the mesh structure: this is captured in SoNICS with operators having a generic number of inputs, that are instantiated with the mesh topology. Figure 3 introduces our running example, which consists of the generation of a very simple operator graph, considering a euler computation of order 1, including a source term, on a mesh with one zone with two border conditions, one of type *Inj* and one of type *wallslip*.

The generation starts with the value modeling the flow update, called `Rhs`, on the only zone of the mesh. One iteration of the loop finds the operator `Rhs` which computes `Rhs` and has the value `Balance` as input. After five other iterations, the operators `FluxBalance`, `SourceTerm1`, `FluxDensity`, `ConvectiveFlux` and `Primitive` were found and inserted in the graph. The next iteration adds the operator `ConvectiveFluxBC` to compute F^c_{BC}: as discussed before, the inputs of this operator depend on the structure of the mesh, and since the mesh's zone has two boundary conditions, this operator has two inputs, one per condition. The rest of the graph generation is straightforward: the two `Conservative` values on the boundary conditions are computed by their respective operators, both depending on the main `Conservative` data. The only value left without an operator computing it is `Conservative`, which is the input of the global computation.

To illustrate how option selection can affect the computation, Figure 4 presents the graph for the same computation, except that its order has been changed to 2. In order 2, the operator

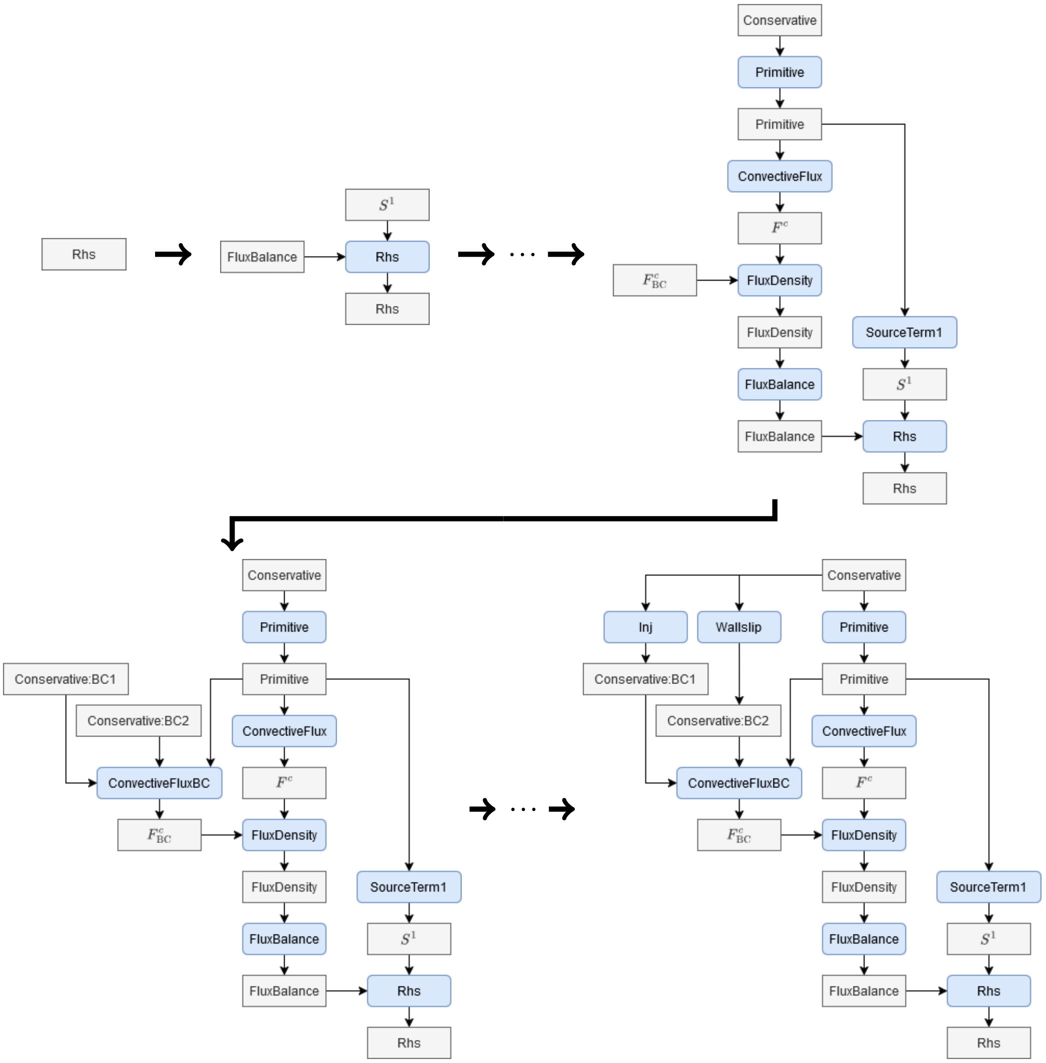


Figure 3. Simple operator graph generation.

`ConvectiveFlux` additionally needs the value `grad(Primitive)` in input. While the rest of the generated graph is identical to the one in Figure 3, the graph generation process includes this new dependency and finds the operator `Gradient` to compute it from the data `Primitive`.

One of the main challenges in creating this automatic graph generation mechanism is to design the DSL in which to express the inputs and outputs of the operators. Since the operators have various properties (e.g., some have a generic number of inputs; others like `Gradient` can produce the gradient of any value), that language must be flexible enough to capture these properties, while still allowing to easily find an operator computing a given value (line 5 of the graph generation algorithm). The details of the language we use in SoNICS are discussed in [40,41].

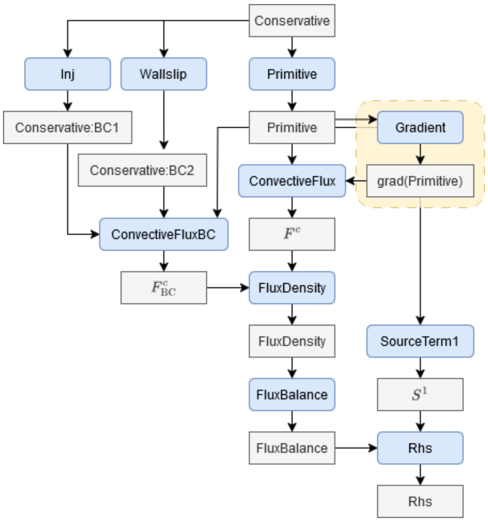


Figure 4. Simple operator graph with order 2.

2.5. *Graph transformation*

These different algorithms are implemented and managed in a similar manner as *passes* during a compilation process: they analyze the operator graph, extract relevant information from it, and modify it w.r.t. this information. The advantage of this architecture is its modularity: (i) every pass has one specific purpose and so like for operators, they are simpler to understand and implement than a complete transformation process, which improves their capability to be checked, tested and maintained; (ii) every pass is independent, and so it is easy to enable, disable or implement new passes depending on the needs. In SoNICS, we have four main passes.

2.5.1. *Distributed computation*

This pass manages the distributed nature of the computation by ensuring that all distant data are computed and adding *data transfer and communication* operations to the graph. Indeed, stencil operations use as input data computed on neighbouring zones: we thus need to ensure that this data is computed, and perform the transfer to make it available for the stencil operations. This pass thus analyses the graph to identify the stencil operators in the graph and the distant data they need, ensure that these data are computed and add the corresponding data transfer operators.

Let us illustrate this pass on the operator graph in Figure 3. The starting point of the pass is the operator graph with some *stencil information*. This stencil information is provided either by the developer (for hand-written operators) or can be automatically extracted from sympy code. Figure 5 presents the stencil information for the operators `Primitive`, `ConvectiveFlux` and `Rhs`. That information gives for every operator the *rank*, i.e., the number of layer of halo cells, required for each of its inputs, and the rank provided for each of its outputs. For instance, Figure 5 states that the `Primitive` operator, having the same rank for its input and output data, does not have a stencil and can be applied on data with a halo of arbitrary size. `ConvectiveFlux` is a stencil operator, since that even if it can be applied on data with a halo of arbitrary size, the halo produced in output is one less than the halo required in input. Finally, `Rhs`, like `Primitive` is not a stencil operation.

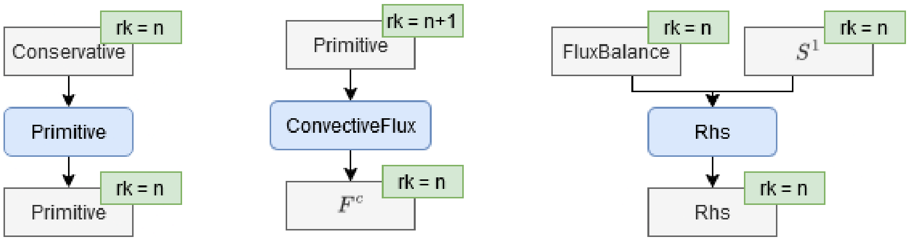
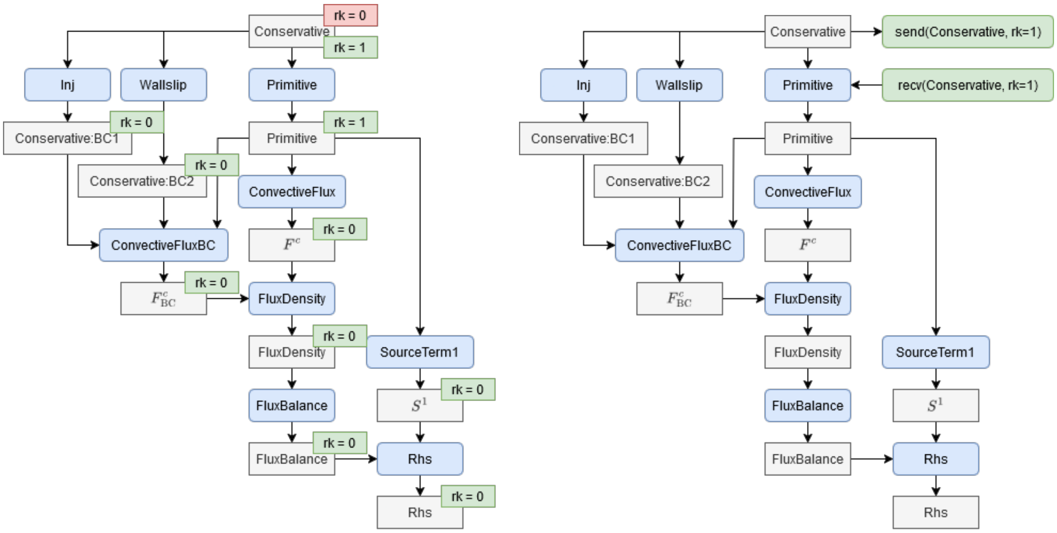


Figure 5. Stencil information of the operators `Primitive`, `ConvectiveFlux` and `Rhs`.



(a) Rank computation

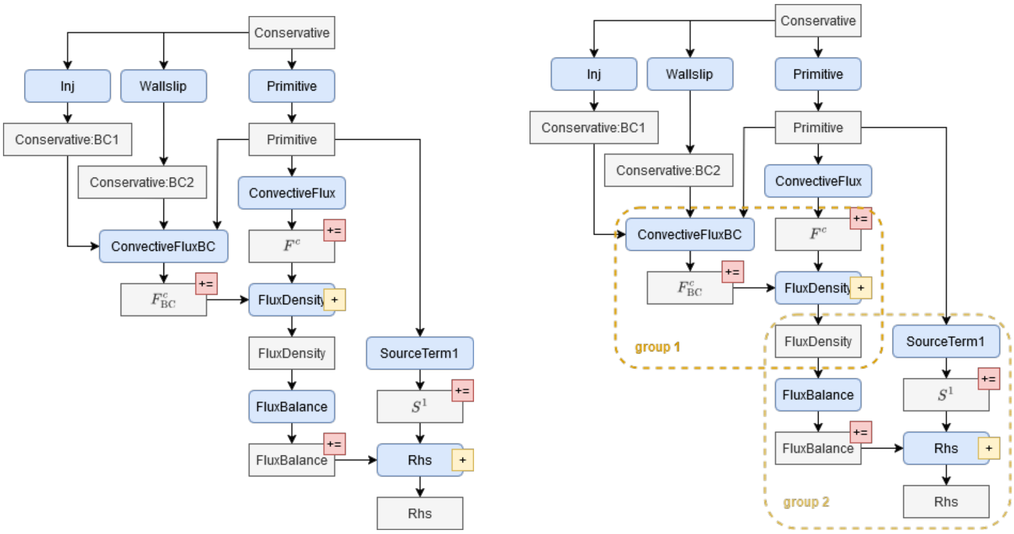
(b) Communicating operator graph

Figure 6. Distributed computation: analysis and transformation of the operator graph.

Using this information, the pass computes the required and provided rank for every data in the operator graph. This computation set the ranks of the inputs and outputs of the graph (i.e., `Rhs` and `Conservative`) and propagate the constraints given by the stencil information in order to minimize the number of data where the provided rank is strictly less than the required rank, i.e., data that would require a communication. Figure 6(a) shows the computed rank for our example graph (if the required and provided rank are the same, only one value is given). Here, almost all data have the same provided and required rank, except `Conservative` which is provided with a rank 0 and required with a rank 1, due to `ConvectiveFlux` being a stencil operation. Considering that the zone is distributed on two computation nodes, this rank discrepancy triggers: (i) ensuring that the other node does provide the `Conservative` data; and (ii) add to the graph the necessary communication operators to ensure that the needed rank is available before the requiring operator is executed. The resulting graph is shown in Figure 6(b): a `send` operation is added to provide the required `Conservative` halo to the other node, and a `recv` operation is added before `Primitive` so it can perform its task on a `Conservative` data with a halo of size 1.

2.5.2. *Legacy optimization*

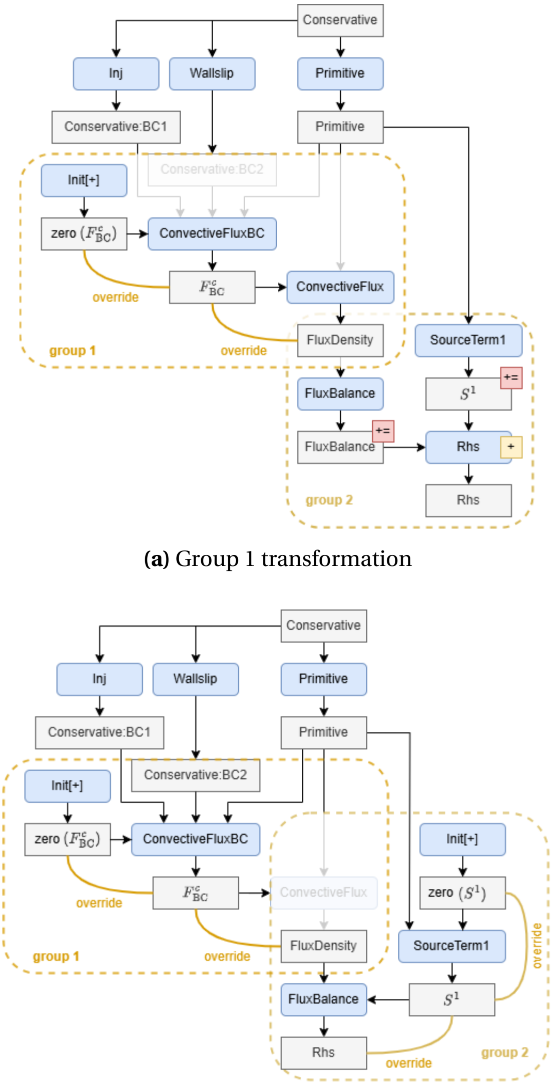
This pass implements a general mechanism to enable a memory and computation optimization that were implemented in *elsA* and is still part of the hand-written operators in SoNICS. This optimization relies on the fact that some operators perform a simple arithmetic operation on their inputs, e.g., in Figure 3, the Rhs and FluxDensity operators simply sum their inputs into their output data. To illustrate the optimization, let us focus on the FluxDensity operator: here, instead of computing F^c_{BC} and F^c independently, storing them into different arrays and then summing them, maybe it is possible to use one unique array initialized to 0 and having directly the ConvectiveFluxBC and ConvectiveFlux increment this array in sequence. This is only possible if these two operators have the possibility to perform these increments. This pass thus implements an analysis to check if this optimization is possible, and then transform the graph in consequence.



(a) Arithmetic operation information **(b)** Group analysis

Figure 7. Legacy optimization: analysing the operator graph.

Let us illustrate this pass on the operator graph in Figure 3. The starting point of the pass is the operator graph with some information stating which operator is performing a simple arithmetic operation, and which operator can integrate this operation into an assignment. Like for the previous pass, this information is provided either by the developer or directly inferred from the operator's implementation when possible. Figure 7(a) presents the information for all the operators in the graph. Here, we can see that the operators FluxDensity and Rhs only perform a sum operation, and the operators ConvectiveFluxBC, ConvectiveFlux, FluxBalance and SourceTerm1 can integrate the sum operation into an assignment. The pass then identifies the groups of operators that could be transformed into a sequence using only one array. Figure 7(b) shows that in our graph, two groups are identified: one containing the operators FluxDensity, ConvectiveFluxBC and ConvectiveFlux; and one containing the operators Rhs, FluxBalance and SourceTerm1. Finally, Figure 8 illustrates how these groups are transformed by the pass. First, Figure 8(a) presents how the group 1 is transformed. First, a new data zero(F^c_{BC}) is added, initialized to zero by the operator Init[+]. This array is added as input to the FluxBC operator which adds F^c_{BC} to it (the fact that the array is modified inplace is modeled with the override edge between the data). Then, this array is given as input to `ConvectiveFlux`, which adds F_c to it: after this sequence, the data contained in this array is equal to $0+F^c_{\mathrm{BC}}+F_c$, i.e., to `FluxDensity`. Consequently, the operator `FluxDensity` is removed from the graph, and our array is directly given as input to `FluxBalance`. In Figure 8(b), the group 2 is transformed in a similar way as the group 1, with a new array `zero(`S^1`)` initialize to 0, first updated by the operator `SourceTerm1`, then updated again by the operator `FluxBalance`, resulting in the sum $0+S^1+$ `FluxBalance`, i.e., the `Rhs` data.



(a) Group 1 transformation

(b) Group 2 transformation

Figure 8. Legacy optimization: transforming the operator graph.

2.5.3. *Graph differentiation*

This pass manages the differentiation of the graph, and since the graph does not contain any complex control flow, it is based on the method developed for the automatic source transformation [12,13] to differentiate the graph, both in forward and backward modes. This method, applied to a graph and considering that the user wants to produce the derivative $\partial D_1/\partial D_2$, can be summed up as the three following steps.

(1) *Operator Identification*: this step identifies every operators used to compute D_1 from D_2, with their inputs and outputs involved in this computation.
(2) *Operator Differentiation*: for all of the identified operators, this step automatically generates (using automatic source transformation [12,13]) a new operator producing the derivative of the identified outputs w.r.t. the identified inputs.
(3) *Assembly*: this step combines these new operators together witht the ones already present in the operator graph to obtain a complete computation of the desired derivative.

Between the forward and backward mode, only the *Operator Differentiation* step changes.

Let us illustrate this pass on the operator graph in Figure 8(b)[1] by requesting the forward derivative ∂`Rhs`/∂`Conservative`. The *Operator Identification* step thus collects all operators between `Rhs` and `Conservative`, i.e., all operators of the graph, with all their inputs and outputs. Then, the *Operator Differentiation* step produces a derivative for every operator in the graph. We distinguish between the forward and the backward mode.

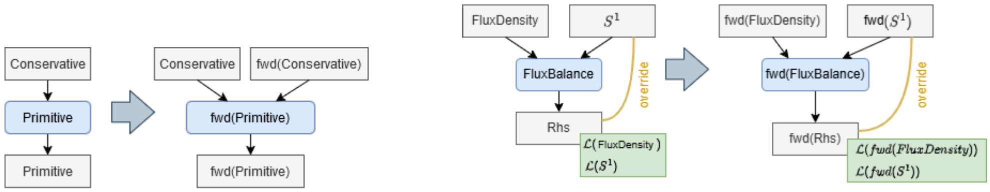
Forward mode. In forward mode, the derivative of an operator can be generated from a set of four rules. If we consider O_d an operator and O_{fwd} its forward derivative, with d_o being any derived output and d_i any deriving input, the rules can be formulated as follows:

(1) the forward derivative of d_o (resp. d_i) must be an output (resp. an input) of O_{fwd};
(2) if d_i is used in a non-linear manner in defining d_o, then d_i must be an input of O_{fwd};
(3) if d_o is stored in the same array as d_i, then the forward derivative of d_o is stored in the same array as the forward derivative of d_i.

Figure 9 illustrates these rules by presenting how the derivative of `Primitive` and `FluxBalance` are produced. First, `Primitive` is an operator taking `Conservative` in input and producing `Primitive` in output, without any additional properties. So its derivative, named `fwd(Primitive)` produces in output the derivative of `Primitive` (called `fwd(Primitive)`) and takes in parameter both the original inputs of `Primitive` (i.e., `Conservative`) and the derivative of these inputs (i.e., `fwd(Conservative)`). The `FluxBalance` operator takes `FluxDensity` and S^1 in input, and produces `Rhs` in output, with the additional property that `Rhs` is stored in the same array as S^1, and that the computation of `Rhs` is linear w.r.t. both of the operator's inputs (noted $\mathscr{L}$(`FluxDensity`), $\mathscr{L}(S^1)$). Since the usage of the operator's inputs is linear, its derivative `fwd(FluxBalance)` does not need the direct values in input, and only uses `fwd(FluxDensity)` and `fwd(`S^1`)` to produce `fwd(Rhs)`. Moreover, the memory usage pattern is the same between the direct operator and its derivative: `fwd(FluxBalance)` uses the array storing `fwd(`S^1`)` to put its `fwd(Rhs)` output.

Finally, the *Assembly* step produces a complete operator graph computing the requested derivative, which is presented in Figure 10. To improve readability, the differentiated operators have a pink color, and edges connecting derivative data are also pink. We can notice that some direct operators present in Figure 8(b) are not present anymore in this graph: this is due to the fact that some operators are linear and so do not need the direct data in input, which thus does not need to be computed.

[1] Since the derivation process is order-sensitive, it must be applied after the pass `Legacy Optimization` which reorders the operators.



(a) Derivative of `Primitive`

(b) Derivative of `FluxBalance`

Figure 9. Graph differentiation: example of operator derivative for forward mode.

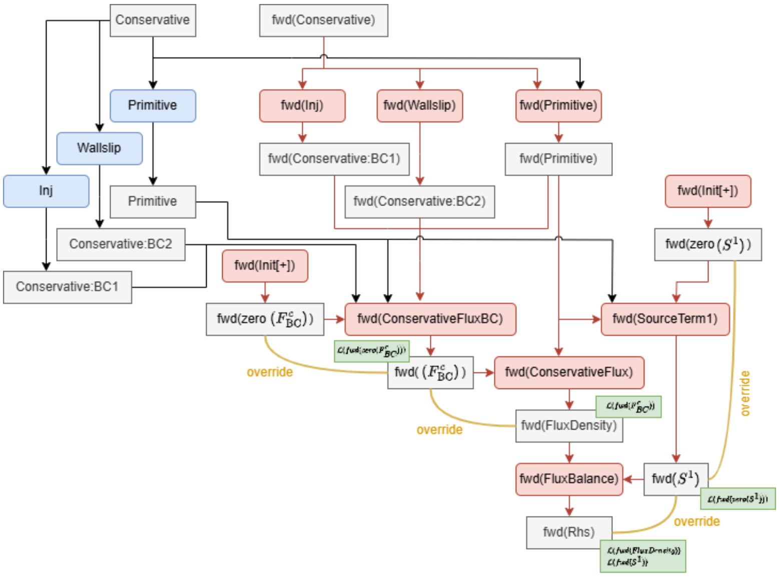


Figure 10. Graph differentiation: computing ∂`Rhs`/∂`Conservative` in forward mode.

Backward mode. In backward mode, the set of rules to produce the derivatives are more complex than in the forward mode and lead to a bigger operator graph. This complexity is due to the fact that the information flow in the backward derivative is reversed w.r.t. the direct graph. Hence, if some data is used by several operators in the direct graph, the backward derivative of this data is constructed by the sum of the contributions of all these operators' backward derivative. To deal with the fact that a direct data can have multiple contributors in backward mode, we tag these backward data with the name of the contributing operator. If we consider O_d an operator and O_{bwd} its backward derivative, with d_o being any derived output and d_i any deriving input, the rules can be formulated as follows:

(1) the backward derivative of d_o (resp. d_i) must be an input (resp. an output) of O_{bwd};
(2) if d_i is used in a non-linear manner in defining d_o, then d_i must be an input of O_{bwd};
(3) if d_o is stored in the same array as d_i, then the backward derivative of d_i is stored in the same array as the backward derivative of d_o;
(4) if d_i is not overridden by any output of O_d, then its backward derivative supports incrementation (see Section 2.5.2).

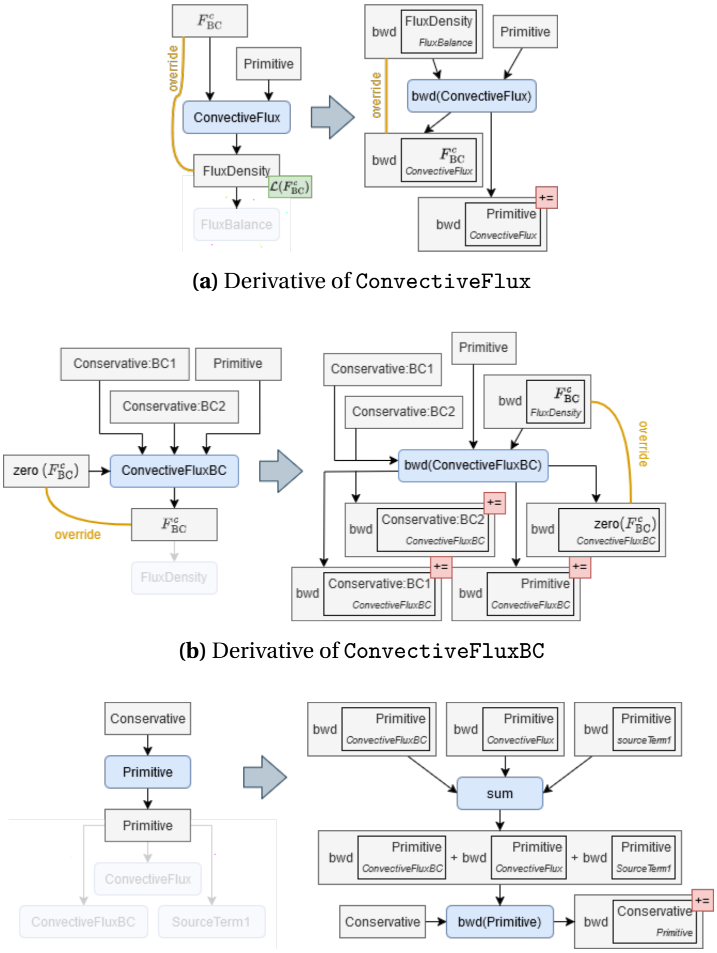
Figure 11 illustrates these rules by presenting how the derivative of `ConvectiveFluxBC`, `ConvectiveFlux` and `Primitive` are produced. First, `ConvectiveFlux` is an operator taking F^c_{BC} and `Primitive` in parameter and returning `FluxDensity` in output, with the additional properties that `FluxDensity` is stored in the same array as F^c_{BC} and is linear w.r.t. the same value. Note that the output `FluxDensity` is used only by the operator `FluxBalance` (which is thus the only contributor to the backward derivative of `FluxBalance`). Consequently, the backward derivative of `ConvectiveFlux`, called `bwd(ConvectiveFlux)`, takes in parameter the backward derivative of `FluxDensity`, called `bwd(FluxDensity)` and tagged only with `FluxBalance`, and also `Primitive` since `FluxDensity` is not linear w.r.t. this data. In output, `bwd(ConvectiveFlux)` produces the backward derivative of F^c_{BC} and `Primitive`, both tagged with `ConvectiveFlux`. Moreover, since `FluxDensity` is stored in the same array as F^c_{BC}, `bwd(`F^c_{BC}`)` is stored in the same array as `bwd(FluxDensity)`; and since `Primitive` is not overridden by any output of `ConvectiveFlux`, `bwd(Primitive)` supports incrementation. The construction of the backward derivative of `ConvectiveFluxBC` is similar, except that since it has more inputs and no linear or overriding properties, the resulting operator has more inputs and outputs, and all its outputs supports incrementation. The construction of the backward derivative of `Primitive` concludes our presentation. The primitive operator has a simple structure, with only one input and one output, with no linear or overriding properties, but its output `Primitive` is used by three different operators that will all become contributors to the backward derivative of the `Primitive` data. Hence the operator `bwd(Primtive)` has only one output, `bwd(Conservative)` which supports incrementation, and two inputs: `Conservative` (since `Primitive` has no linear properties), and the main one that collects in a sum all the contributions made to `bwd(Primitive)`.

Finally, the *Assembly* step produces a complete operator graph computing the requested derivative, which is presented in Figure 12 where direct operators have been removed for readability.

2.5.4. *Memory management*

This pass completes the graph with *memory allocation* and *array copy* operators. Indeed, all the data computed in the graph must be stored on some array which must be allocated. Some of these array must be allocated only once (for data whose size do not change over time), some other must be reallocated regularly, e.g., for iso-surfaces. Moreover, as illustrated in the *Legacy Optimization* pass, some operator may override the content of an array with its own output, possibly creating issues if that content is used by another operator. To avoid such data-race, in most case it is enough to force the execution of the overriding operator after the other ones using the data. But in rare cases, the input data must be copied. This pass thus analyses the data manipulated by the graph, its size, and computes an allocation scheme suited for it, i.e., it associates a memory space in an array for every data while ensuring that data is overwritten only when it is not used anymore. Then it adds to the graph the corresponding memory allocation operators.

Figure 13 presents the result of the pass on the graph in Figure 8(b). Since every overwritten data is used by only one operator, no ordering between operators nor copies were added to the graph. However, five allocation operators were added to the graph: (i) the `Primitive` data is created by the `Primitive` operator (which needs an array to store it) and so that array is allocated by the operator `Alloc[Primitive]` before the `Primitive` operator is executed; (ii) and (iii) similarly for the `Inj` and `Wallslip` operators, the array where they store their output is allocated before they execute; (iv) since the data `zero(`F^c_{BC}`)`, F^c_{BC} and `FluxDensity` all share the same array, only one allocation operator is added to create the array that will be filled with 0 by the `Init[+]` operator on the left of the graph; and (v) similarly with the data `zero(`S^1`)`, S^1 and `Rhs`, only one allocation operator is added to create the array that will be filled with 0 by the



(a) Derivative of `ConvectiveFlux`

(b) Derivative of `ConvectiveFluxBC`

(c) Derivative of `Primitive`

Figure 11. Graph differentiation: example of operator derivative for backward mode.

`Init[+]` operator on the right of the graph. Note that since the `Conservative` data is an input of the graph, it does not need to be allocated.

3. Results

SoNICS and its architecture have been gradually tested and validated from simple cases like NACA012 airfoil on inviscid flow [42,43] to more complex academical case like the ONERA M6 Wing [44] and specific open usecases to validate turbulence models [45], up to industrial cases. Currently, SoNICS supports: Conducted Reynolds-Averaged Navier–Stokes (RANS) simulations with Spalart–Allmaras [46] and k-ω [47] turbulence models; Incorporated transition modeling through transport equations, specifically employing the Menter–Langtry transition model [48]; and Multi-species simulations to account for combustion phenomena. Moreover, unsteady computation with Runge–Kutta [49] or BDF2 [50] approaches and advanced version of the Zonal Detached Eddy Simulation (ZDES) based on Spalart–Allmaras model proposed by Deck [51] have also been implemented, tested and validated. Furthermore, to effectively manage computations

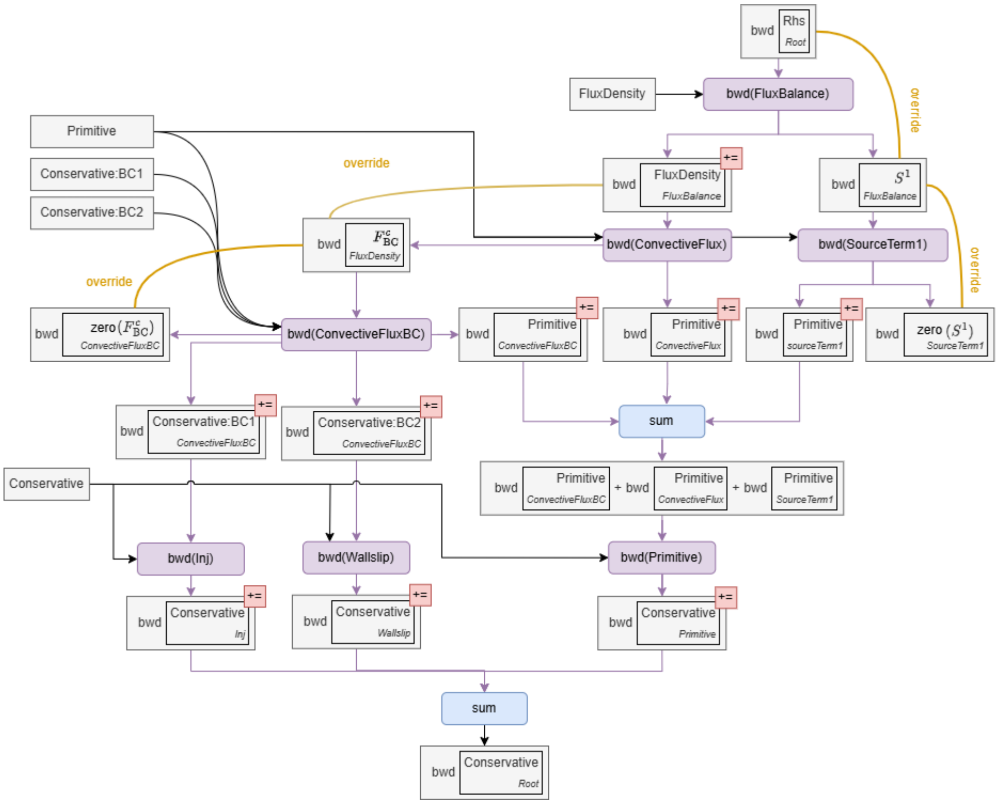


Figure 12. Graph differentiation: computing ∂`Rhs`$/\partial$`Conservative` in backward mode.

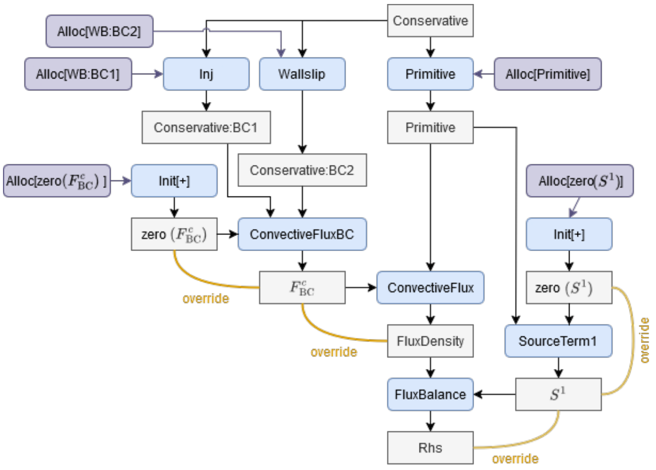


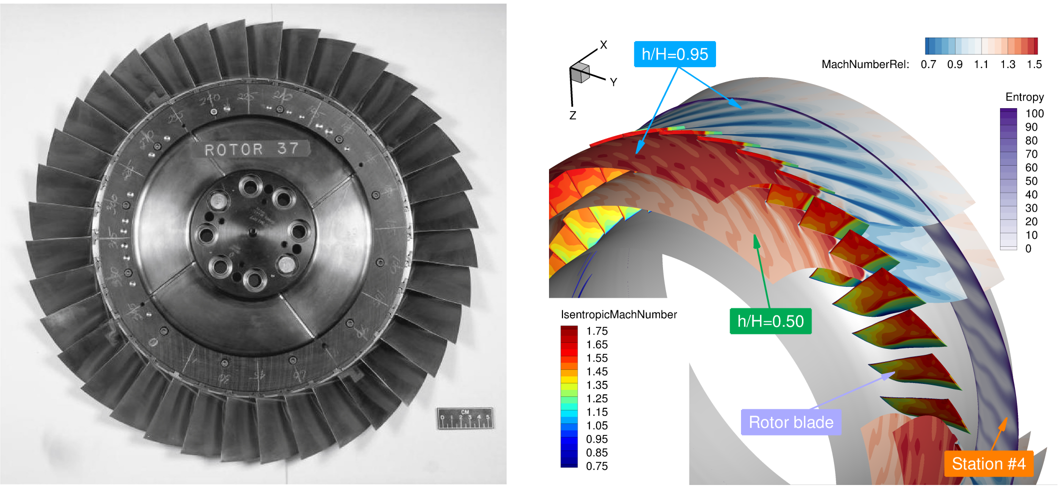
Figure 13. Memory management: adding the allocation operators.

on meshes with high anisotropy ratios, SoNICS includes a vertex-centered solver. This approach is widely recognized for its robustness when dealing with highly anisotropic meshes, ensuring accurate and stable numerical solutions even in challenging geometric configurations.

The rest of the section presents two cases of interest to highlight the different architecture choices of SoNICS: (i) the NASA Rotor 37 [52] demonstrates the capabilities of turbomachinery simulations and compares the results of SoNICS to those of *elsA*; (ii) a generic rocket after body [53,54] compares the cell-centered solver and the vertex-centered approach with mesh adaptation.

3.1. *NASA Rotor 37*

The NASA Rotor 37, presented in Figure 14, is a widely recognized and extensively studied test case in the CFD field for turbomachinery applications. This transonic axial compressor rotor was designed in the 1970s and was tested at NASA's Lewis Research Center (now Glenn Research Center) [55]. As an isolated rotor, the NASA Rotor 37 continues to be relevant in modern CFD research, serving as a platform for testing new numerical methods, turbulence models, and optimization techniques. Its well-documented geometry and experimental data make it an ideal candidate for validating CFD solvers and exploring advanced concepts in turbomachinery flow physics.



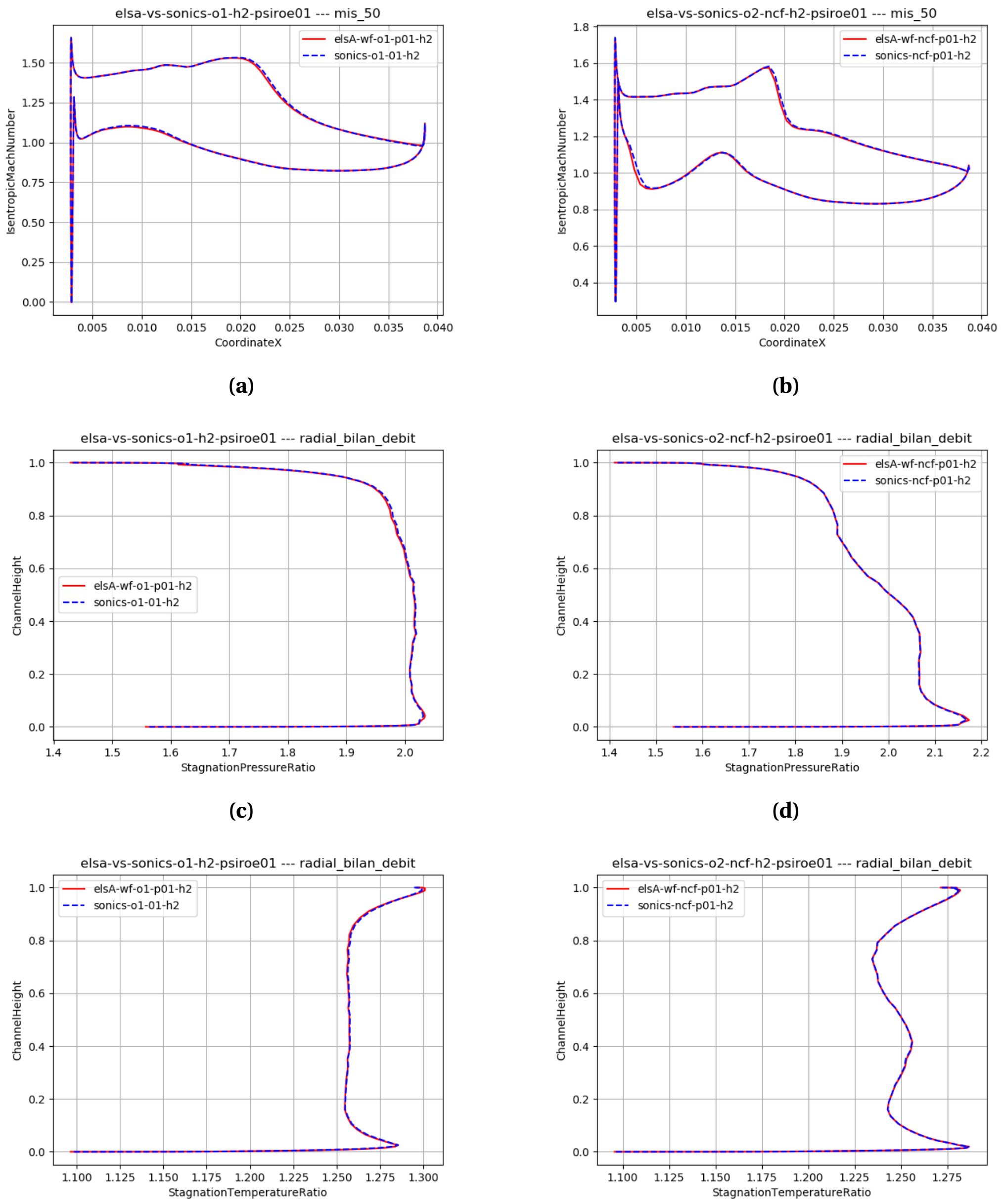
(a) NASA Rotor 37 blade

(b) Computation with SoNICS

Figure 14. NASA Rotor 37.

To validate SoNICS, a comparative study is conducted using the NASA Rotor 37. The same calculation is performed using both *elsA* and SoNICS in a nearly identical numerical configuration using a Roe scheme on an unstructured hexahedral mesh consisting of 1,480,704 cells. The turbulence is modeled using the Spalart–Allmaras (SA) model, a widely adopted one-equation Reynolds-Averaged Navier–Stokes (RANS) turbulence model. Our comparison focuses on several key parameters of interest, including the isentropic Mach number. Moreover, to showcase the flexibility of SoNICS's architecture, the comparison is carried out on two different scenarios: (i) first-order accuracy; and (ii) second-order accuracy without slope limiters.

Figure 15 shows respectively a slice of the isentropic Mach number at mid span blade (h/H = 50%), the distribution of the stagnation pressure and temperature ratio among the radius at station 4 (cf. Figure 14(b)). We can see here that the results between *elsA* and SoNICS are similar: the tiny differences are caused by the fact that turbulence model are not strictly the same (the model in SoNICS slightly improves over the one in *elsA* by taking into account compressibility phenomena).



(a)

(b)

(c)

(d)

(e)

(f)

Figure 15. elsA and SoNICS comparison of isentropic Mach number (a) at order 1 and (b) order 2; stagnation pressure ratio (c) at order 1 and (d) order 2; stagnation temperature ratio (e) at order 1 and (f) order 2.

3.2. *Simplified launcher afterbody*

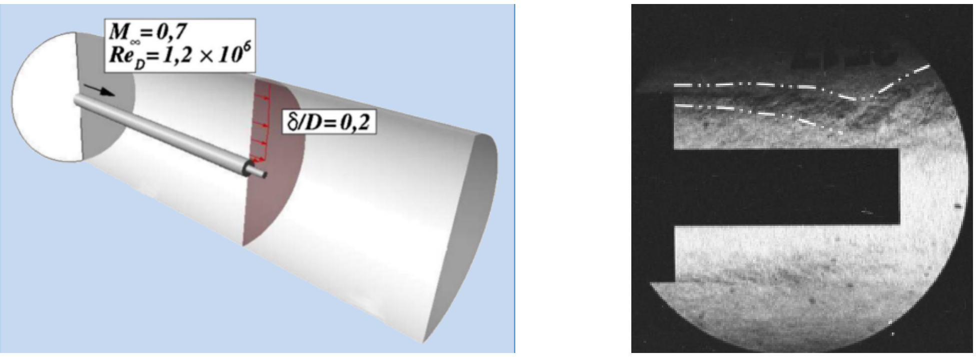
This case consists of two concentric cylinders with the inner cylinder modeling the nozzle as shown in Figure 16. The primary cylinder has a diameter of $D = 0.1\,\mathrm{m}$ and a length of $L = 1.2D$.

Experimental investigations were conducted in ONERA's S3Ch transonic wind tunnel [53] and advanced turbulence modelisation like Zonal Detached Eddy Simulation (ZDES) approach have been studied at ONERA on this case [54]. The experiment was conducted at a Mach number of $M = 0.7$ with a Reynolds number based on the cylinder diameter of $\mathrm{Re}_D = 1.2 \times 10^6$ and such that the boundary layer thickness at the axisymmetric step is $\delta = 0.2D$. The Reynolds number is sufficiently high to justify the use of Reynolds-Averaged Navier–Stokes (RANS) modeling for turbulence. In this particular case, the Spalart–Allmaras model is employed as the turbulence closure scheme.

Two computations have been done on this geometry:

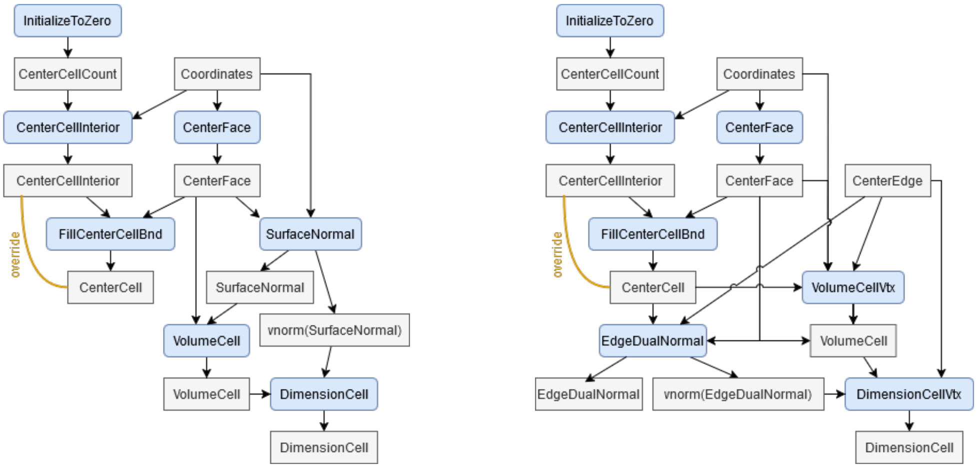
(1) a simulation performed with the cell-center solver on a fine hand-made hexa mesh;

(2) a simulation performed with the vertex-center solver on tetra meshes; the simulation is initiated on an extremely coarse tetrahedral mesh that was refined using a local Mach number criterion; this adaptive mesh refinement procedure was iteratively repeated until a predefined convergence criterion was satisfied.



(a) Overview extracted from [54] **(b)** Strioscopy extracted from [56]

Figure 16. Generic rocket aft body S3Ch.



(a) Cell-center solver **(b)** Vertex-center solver

Figure 17. The geometry part of the operator graph.

Figure 17 shows the flexibility of SoNICS's architecture by presenting the part of the operator graph dedicated to the computation of the geometry for both computations: while these graphs are very different, they were automatically produced and executed by SoNICS's architecture alone, with one option changed it input. Figure 18 shows the remarkable superposition between the isolines obtained from the cell-centered solver simulation and the contours resulting from the vertex-centered solver simulation after 15 adaptive refinement iterations. And finally, Figure 19 presents a comparative analysis of the pressure coefficient distributions obtained from two sources: (i) the cell-centered solver simulation; and (ii) the vertex-centered solver simulation after 15 adaptive refinement iterations. This validates the precision and reliability of the vertex-center solver w.r.t. the cell-center solver inherited from *elsA*, and moreover, not shown in this figure is the high degree of consistency between both numerical methods in this test case.

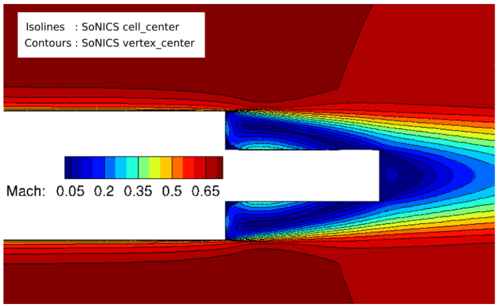


Figure 18. Isolines from the cell-centered solver simulation, superposed with the contours from the vertex-centered solver simulation after 15 adaptive refinement iterations: they are identical.

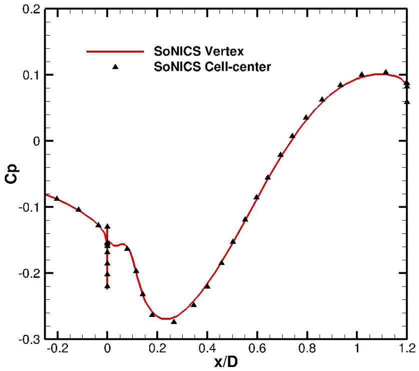


Figure 19. Pressure coefficient distributions from the cell-centered solver, and the vertex-centered solver after 15 adaptive refinement iterations.

4. Related work

Many CFD solvers have been developed over the years. However, while plenty of bibliography exists on the models supported by these tools, their simulations and performances, very little can be found on their workflow, even for open source solvers. In the following, we present five of the most well-known solvers.

AVBP. This software [57,58] is developed at CERFACS and is specifically designed for simulating unsteady turbulent compressible and reactive flows and especially Large Eddy Simulation (LES). It is capable of massive parallel processing, supporting both CPU and GPU computations, and is implemented in Fortran.

No documentation on its workflow is directly available, but the available presentations of the tool indicate that it does not separate the control flow from its computation, which can add a large runtime overhead to the computation, as discussed in this article. Its user interface consists of a graphical interface named C3S to facilitate workflow management, and a component called HIP to handle preprocessing. Finally, since AVBP is dedicated to unsteady computation, gradient optimization does not fall within its primary area of interest, and so it does not support automatic code differentiation.

YALES2. This software [59] is developed at CORIA for simulating unsteady, turbulent, incompressible, compressible and reactive flows featuring feature-based mesh adaptation. It is composed by several solvers, each dedicated to a specific family of simulation. All of its solvers are implemented in Fortran 2008, have distributed hardware support, and some have been ported to NVIDIA and AMD GPUs.

No documentation on its workflow is directly available, but it is clear from its design that the part of the control flow dedicated to choosing the simulation model is separated from the computation, since every supported model is implemented by a dedicated solver. Moreover, the control flow dedicated to the management of the hardware and mesh topologies is most probably implemented in a similar fashion as *elsA*, using a factory identifying what to compute before actually starting the simulation. Its user interface consists of a python API that utilizes the f90wrap library to facilitate the various Fortran data types wrapping. Finally, YALES2 is dedicated to unsteady computation and does not support automatic code differentiation.

CODA. This software [60] (previously called *Flucs*) is a collaboration between ONERA, DLR and Airbus and is implemented in C++, using templates for performance optimization. Like *elsA* and SoNICS, it supports a wide range of physical models including (but not limited to): steady-state Reynolds-Averaged Navier–Stokes equations (RANS) with various turbulence models, Large Eddy Simulation (LES), Hybrid RANS/LES models for complex flow scenarios, Multi-species and reactive flows. One noticeable particularity is that CODA implements a second-order finite-volume method and higher-order Discontinuous Galerkin (DG) methods tailored on unstructured grids. Moreover, it supports distributed hardware (based on the MPI [61], GASPI [62] and OpenMP [63] libraries), and while it does not completely support GPUs, its implicit phase is based on the Spliss library [64] which has full GPU support.

The control flow of CODA is split in two parts: the model and schema are managed during compilation, producing one solver for a dedicated family of simulations; and before performing the simulation, a factory identifies what to compute from the hardware and mesh topologies. As user interface, it provides a python API. Automatic Differentiation is implemented in CODA with ADOL-C [65] which performs a dynamic analysis during the simulation, with a possible memory overhead as discussed in [20].

OpenFOAM. This software [66] is a free, open-source solver suite with a modular architecture similar to YALES2 and CODA. It supports a wide range of physical models implemented by around 100 different solvers, and can be executed on distributed hardware, including GPUs [67]. One noticeable particularity is that OpenFOAM's architecture allows for easy coupling between its different solvers.

The control flow of OpenFOAM is split in two parts: the model and schema are managed during compilation, producing one solver for a dedicated family of simulations; and before performing the simulation, a factory identifies what to compute from the hardware and mesh topologies. Its user interface is structured as multiple configuration files that the user need to fill using a dedicated syntax. Finally, a specialized and partial version of OpenFOAM, called DAFoam [68], uses dynamic analysis to achieve Automatic Differentiation based on CoDiPack [16] and MeDiPack [69].

SU2. This software [70] is a free, open-source solver suite implemented in C++ dedicated to diverse fluid dynamics problems, including steady and unsteady simulations, compressible and incompressible flows, turbulence modeling and multi-physics problems. All of its solvers can be run on distributed hardware, but support for GPUs is still partial. SU2's code base was designed for easy customization and integration of new features.

The control flow of SU2 is split in two parts, like the one of OpenFOAM. A python-based graphical interface, called SU2GUI [71] is provided to the user to configure and manage SU2. Finally, Automatic Differentiation is implemented in SU2 with dynamic analysis using CoDiPack.

Although many articles around these different solvers emphasize numerical methods or different physical modeling, information that directly addresses the architectural details of these solvers is difficult to find. However, as discussed for each solver individually, it seems that like *elsA*, they all offer an architecture where user choices are directly transformed into a static list of functions to be executed at runtime. The particularity of SoNICS's architecture is its capability to first generate a graph of operators from the user choices prior to the code execution. This graph can be analyzed and transformed to optimize the runtime execution, similarly to a compiler analysing and optimizing the code in input to produce an efficient executable. Moreover, thanks to SoNICS's code generation capabilities, the implementation of every variant of each operator can be also generated on the fly.

It is important to emphasize that SoNICS's operator graph, its manipulation and distribution are largely inspired by Artificial Intelligence (AI) frameworks like TensorFlow [72] and PyTorch [73]. This alignment is a deliberate choice driven by shared constraints and objectives, like efficiency and portablity. Moreover, modern AI frameworks include Automatic Differentiation capabilities, typically implemented through sophisticated source code transformation tools like Enzyme [13].

5. Conclusion

In this article, we presented the design choices and the structure of the new SoNICS CFD solver. All of these choices were carefully made to satisfy the different expectations a user could have. We illustrated our design with several examples to show how a user configuration is translated, step by step, into an operator graph containing all the information necessary to perform a computation answering the user's request, with a focus on memory and time efficiency. Moreover, we applied the complete tool on two important usecases, the *Rotor 37* and the *Simplified launcher afterbody*, validating its capability to seamlessly manage complex configurations and different hardwares to perform both a direct simulation and a mesh refinement task.

In future work, we intend to improve and add several elements of the workflow. For instance, the `Graph Differentiation` pass (in Section 2.5.3) is still in development; the `Memory Management` pass (in Section 2.5.4) could be improved so more arrays could be reused, thus improving the memory footprint of SoNICS; the `Builder` task (in Section 2.2) can be refactored; and the `Distribution` task (in Section 2.2) could be changed to compute a better load-balancing, using for instance algorithms of the HEFT family [74–76]. Moreover, more memory and time optimizations can be added to the SoNICS's workflow: for instance, operators looping on the same data could be merged if they execute on CPU, so they would share the same loop, thus reducing the latency due to cache misses. Finally, we also intend to perform more simulations using SoNICS, including coupling it with other solvers, to check if its current API is satisfactory or should be improved to satisfy every user's needs.

Acknowledgments

The authors would like to thank Julien Husson, Julien Marty and Bruno Maugars for their work on the usecases presented in this article.

Declaration of interests

The authors do not work for, advise, own shares in, or receive funds from any organization that could benefit from this article, and have declared no affiliations other than their research organizations.

References

[1] L. Cambier, M. Gazaix, S. Heib, S. Plot, M. Poinot, J. P. Veuillot, J.-F. Boussuge and M. Montagnac, "An overview of the multi-purpose *elsA* flow solver", *Aerospace Lab* (2011), no. 2, article no. AL02-10 (15 pages).

[2] L. Cambier, S. Heib and S. Plot, "The Onera elsA CFD software: input from research and feedback from industry", *Mech. Ind.* **14** (2013), no. 3, pp. 159–174.

[3] S. Plot, "The High Level of Maturity of the elsA CFD Software for Aerodynamics Applications", in *EUCASS 2019*, 2019.

[4] W. Thollet, G. Dufour, X. Carbonneau and F. Blanc, "Body-force modeling for aerodynamic analysis of air intake–fan interactions", *Int. J. Numer. Methods Heat Fluid Flow* **26** (2016), no. 7, pp. 2048–2065.

[5] G. Carrier, D. Destarac, A. Dumont, et al., "Gradient-based aerodynamic optimization with the elsA software", in *52nd Aerospace Sciences Meeting*, American Institute of Aeronautics and Astronautics, 2014.

[6] B. G. van Der Wall, C. Kessler, Y. Delrieux, P. Beaumier, M. Gervais, J. C. Hirsch, K. Pengel and P. Crozier, "From aeroacoustics basic research to a modern low-noise rotor blade", *J. Am. Helicopter Soc.* **62** (2017), no. 4, pp. 1–16.

[7] E. R. Leon, A. Le Pape, M. Costes, J. A. Désidéri and D. Alfano, "Concurrent aerodynamic optimization of rotor blades using a Nash game method", *J. Am. Helicopter Soc.* **61** (2016), no. 2, pp. 1–13.

[8] D. Guegan, M. Schvallinger, F. Julienne, N. Gourdain and M. Gazaix, "Three-dimensional full annulus unsteady RANS simulation of an integrated propulsion system", in *51st AIAA/SAE/ASEE Joint Propulsion Conference*, American Institute of Aeronautics and Astronautics, 2015.

[9] T. Berthelon, A. Dugeai, J. Langridge and F. Thouverez, "Ground effect on fan forced response", in *Proc. of the 15th International Symposium on Unsteady Aerodynamics, Aeroacoustics and Aeroelasticity of Turbomachines, Vol. ISUAAAT15-094, American Society of Mechanical Engineers*, 2018.

[10] F. Blondel, M. Stanciu, F. Lebœuf and M. Lance, "Modelling unsteadinesses and polydispersion in wet steam flows using the quadrature method of moments and a two-equation model", in *Proceedings of the 10th European Conference on Turbomachinery Fluid dynamics & Thermodynamics*, 2013.

[11] M. Stanciu, M. Marcelet and J.-M. Dorey, "Numerical Investigation of Condenser Pressure Effect on Last Stage Operation of Low Pressure Wet Steam Turbines", in *ASME Turbo Expo 2013: Turbine Technical Conference and Exposition. Volume 5B: Oil and Gas Applications; Steam Turbines*, ASME Press, 2013, pp. 1–11.

[12] L. Hascoët and V. Pascual, "The Tapenade automatic differentiation tool: Principles, model, and specification", *ACM Trans. Math. Softw.* **39** (2013), no. 3, article no. 20 (43 pages).

[13] W. S. Moses, S. H. K. Narayanan, L. Paehler, V. Churavy, M. Schanen, J. Hückelheim, J. Doerfert and P. Hovland, "Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Press: Dallas, Texas, 2022, (18 pages).

[14] D. Maclaurin, D. Duvenaud and R. P. Adams, "Autograd: effortless gradients in numpy", in *ICML 2015 AutoML Workshop*, 2015.

[15] J. Bradbury, R. Frostig, P. Hawkins, et al., *JAX: composable transformations of Python+NumPy programs*, version 0.3.13, 2018. Online at http://github.com/jax-ml/jax (accessed on October 31, 2025).

[16] M. Sagebaum, T. Albring and N. R. Gauger, "High-Performance Derivative Computations using CoDiPack", *ACM Trans. Math. Softw.* **45** (2019), no. 4, article no. 38 (26 pages).

[17] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic and I. Stoica, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning", in *CGO '20: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (J. Mars, L. Tang, J. Xue and P. Wu, eds.), ACM Press: San Diego, CA, USA, 2020, pp. 242–255.

[18] Y. Chen, C. Mendis and S. Amarasinghe, "All you need is superword-level parallelism: systematic control-flow vectorization with SLP", in *PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (R. Jhala and I. Dillig, eds.), ACM Press: San Diego, CA, USA, 2022, pp. 301–315.

[19] A. H. Tabar, R. Bubel and R. Hähnle, "Automatic loop invariant generation for data dependence analysis", in *FormaliSE '22: Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering* (S. Gnesi, N. Plat, A. Hartmanns and I. Schaefer, eds.), ACM Press, 2022, pp. 34–45.

[20] B. Maugars, S. Bourasseau, C. Content, B. Michel, B. Berthoul, J. N. Ramirez, P. Raud and L. Hascoët, "Algorithmic Differentiation for an efficient CFD solver", in *ECCOMAS 2022: 8th European Congress on Computational Methods in Applied Sciences and Engineering*, 2022.

[21] P. Clements and L. Northrop, *Software product lines: practices and patterns*, Addison Wesley Longman, 2001.

[22] D. Poirier, S. R. Allmaras, D. McCarthy, M. Smith and F. Enomoto, "The CGNS system", in *29th AIAA, Fluid Dynamics Conference*, American Institute of Aeronautics and Astronautics, 1998.

[23] M. Poinot and C. L. Rumsey, "Seven keys for practical understanding and use of CGNS", in *2018 AIAA Aerospace Sciences Meeting*, American Institute of Aeronautics and Astronautics, 2018.

[24] J. Coulet, C. Benazet, B. Berthoul and B. Maugars, *Maia: a Python and C++ library for parallel algorithms and manipulations over CGNS meshes*, version 1.5, 2024. Online at https://github.com/onera/maia/ (accessed on October 31, 2025).

[25] G. Kiczales, "Aspect-oriented programming", *ACM Comput. Surv.* **28** (1996), no. 4es, article no. 154.

[26] G. Kiczales, "AspectJ(tm): Aspect-Oriented Programming in Java", in *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers* (M. Aksit, M. Mezini and R. Unland, eds.), Lecture Notes in Computer Science, Springer, 2002, p. 1.

[27] G. Kiczales, "Aspect-oriented programming", in *ICSE '05: Proceedings of the 27th International Conference on Software Engineering* (G.-C. Roman, W. G. Griswold and B. Nuseibeh, eds.), ACM Press, 2005, p. 730.

[28] F. Pellegrini and J. Roman, "SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs", in *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1996, Brussels, Belgium, April 15-19, 1996, Proceedings* (H. M. Liddell, A. Colbrook, L. O. Hertzberger and P. M. A. Sloot, eds.), Lecture Notes in Computer Science, Springer, 1996, pp. 493–498.

[29] R. Barat, C. Chevalier and F. Pellegrini, "Multi-criteria Graph Partitioning with Scotch", in *Proceedings of the Eighth SIAM Workshop on Combinatorial Scientific Computing, CSC 2018, Bergen, Norway, June 6-8, 2018* (F. Manne, P. Sanders and S. Toledo, eds.), Society for Industrial and Applied Mathematics, 2018, pp. 66–75.

[30] G. Karypis, K. Schloegel and V. Kumar, "ParMETIS — Parallel graph partitioning and sparse matrix ordering library", version 3.1, 2003. Online at http://charm.cs.uiuc.edu/users/gupta/2012_CloudHPCLB_charm/src/libs/ck-libs/parmetis/Manual/manual.pdf.

[31] G. Karypis, "METIS and ParMETIS", in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), Springer, 2011, pp. 1117–1124.

[32] E. Quemerais, B. Maugars and B. Andrieu, *ParaDiGM: a library for parallel computational geometry*, version 2.6.0, 2024. Online at https://github.com/onera/paradigm/ (accessed on October 31, 2025).

[33] T.-W. Huang, D.-L. Lin, C.-X. Lin and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System", *IEEE Trans. Parallel Distrib. Syst.* **33** (2022), no. 6, pp. 1303–1320.

[34] D. Romero-Organvidez, P. N. Ayuso, J. A. Galindo and D. Benavides, "Kconfig metamodel: a first approach", in *Proceedings of the 28th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2024, Dommeldange, Luxembourg, September 2-6, 2024* (M. Cordy, D. Strüber, M. Pinto, et al., eds.), ACM Press, 2024, pp. 55–60.

[35] R. Schröter, S. Krieter, T. Thüm, F. Benduhn and G. Saake, "Feature-model interfaces: the highway to compositional analyses of highly-configurable systems", in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser and L. A. Williams, eds.), ACM Press, 2016, pp. 667–678.

[36] I. Schaefer, L. Bettini, V. Bono, F. Damiani and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines", in *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings* (J. Bosch and J. Lee, eds.), Lecture Notes in Computer Science, Springer, 2010, pp. 77–91.

[37] M. Lienhardt, "PYDOP: A Generic Python Library for Delta-Oriented Programming", in *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume B, SPLC 2023, Tokyo, Japan, 28 August 2023- 1 September 2023* (P. Arcaini, M. H. t. Beek, G. Perrouin, et al., eds.), ACM Press, 2023, pp. 30–33.

[38] A. Meurer, C. P. Smith, M. Paprocki, et al., "SymPy: symbolic computing in Python", *PeerJ Comput. Sci.* **3** (2017), article no. e103 (27 pages).

[39] M. Bauer, J. Hötzer, D. Ernst, et al., "Code generation for massively parallel phase-field simulations", in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019* (M. Taufer, P. Balaji and A. J. Peña, eds.), ACM Press, 2019, (32 pages).

[40] M. Lienhardt, M. H. t. Beek and F. Damiani, "Product lines of dataflows", *J. Syst. Software* **210** (2024), article no. 111928 (22 pages).

[41] M. Lienhardt, "The Hrewrite Library: A Term Rewriting Engine for Automatic Code Assembly", in *Rewriting Logic and Its Applications - 15th International Workshop, WRLA 2024, Luxembourg City, Luxembourg, April 6-7, 2024, Revised Selected Papers* (K. Ogata and N. Martí-Oliet, eds.), Lecture Notes in Computer Science, Springer, 2024, pp. 165–178.

[42] K. W. McAlister, L. W. Carr and W. J. McCroskey, *Dynamic stall experiments on the NACA 0012 airfoil*, Technical Publication, NASA, no. 19780009057, 1978.

[43] W. J. McCroskey, *A critical assessment of wind tunnel results for the NACA 0012 airfoil*, Technical Memorandum, NASA, no. 19880002254, 1987.

[44] V. Schmitt and F. Charpin, *Pressure distributions on the ONERA-M6-Wing at transonic Mach numbers*, techreport, ONERA, no. AGARD AR 138, 1979.

[45] Turbulence Model Benchmarking Working Group, *Turbulence Modeling Resource*, June 3, 2025. Online at https://turbmodels.larc.nasa.gov/ (accessed on October 30, 2025).

[46] P. R. Spalart and S. R. Allmaras, "A one-equation turbulence model for aerodynamic flows", *Rech. Aerosp.* **42** (1994), no. 1, pp. 5–21.

[47] F. R. Menter, "Two-Equation Eddy-Viscosity Turbulence Models for Engineering Applications", *AIAA J.* **32** (1994), no. 8, pp. 1598–1605.

[48] R. B. Langtry and F. R. Menter, "Correlation-Based Transition Modeling for Unstructured Parallelized Computational Fluid Dynamics Codes.", *AIAA J.* **47** (2009), no. 12, pp. 2894–2906.

[49] J. H. Williamson, "Low-Storage Runge-Kutta Schemes.", *J. Comput. Phys.* **35** (1980), pp. 48–56.

[50] C. W. Gear, *Numerical initial value problems in ordinary differential equations*, Prentice Hall, 1971.

[51] S. Deck, "Recent improvements in the Zonal Detached Eddy Simulation (ZDES) formulation", *Theor. Comput. Fluid Dyn.* **26** (2012), pp. 523–550.

[52] S. Mouriaux, F. Bassi, A. Colombo and A. Ghidoni, "NASA Rotor 37", in *TILDA: Towards Industrial LES/DNS in Aeronautics: Paving the Way for Future Accurate CFD - Results of the H2020 Research Project TILDA, Funded by the European Union, 2015 -2018* (C. Hirsch, K. Hillewaert, R. Hartmann, V. Couaillier, J.-F. Boussuge, F. Chalot, S. Bosniakov and W. Haase, eds.), Springer, 2021, pp. 533–544.

[53] D. Deprés, P. Reijasse and J. P. Dussauge, "Analysis of unsteadiness in afterbody transonic flows", *AIAA J.* **42** (2004), no. 12, pp. 2541–2550.

[54] R. Pain, P.-E. Weiss and S. Deck, "Zonal Detached Eddy Simulation of the Flow Around a Simplified Launcher Afterbody", *AIAA J.* **52** (2014), no. 9, pp. 1967–1979.

[55] L. Reid and R. D. Moore, *Design and overall performance of four highly loaded, high-speed inlet stages for an advanced high-pressure-ratio core compressor*, Technical Paper, NASA, no. NASA-TP-1337, 1978.

[56] P. Reijasse, *Étude expérimentale de l'écoulement d'arrière-corps du lanceur Ariane 5 dans la soufflerie transsonique S3Ch*, techreport, ONERA, 2007.

[57] CERFACS, *AVBP*, version 7.15, 2025. Online at https://avbp-wiki.cerfacs.fr/ (accessed on October 31, 2025).

[58] T. Schonfeld and M. Rudgyard, "Steady and unsteady flow simulations using the hybrid flow solver AVBP", *AIAA J.* **37** (1999), no. 11, pp. 1378–1385.

[59] V. Moureau, P. Domingo and L. Vervisch, "Design of a massively parallel CFD code for complex geometries", *Comptes Rendus. Mécanique* **339** (2011), no. 2–3, pp. 141–148.

[60] T. Leicht, J. Jägerskupper, D. Vollmer, A. Schwöppe, R. Hartmann, J. Fiedler and T. Schlauch, "DLR-Project Digital-X — Next Generation CFD Solver 'Flucs'", in *Deutscher Luft- und Raumfahrtkongress 2016*, 2016, (14 pages).

[61] F. Nielsen, "Introduction to MPI: The Message Passing Interface", in *Introduction to HPC with MPI for Data Science*, Undergraduate Topics in Computer Science, Springer, 2016, pp. 21–62.

[62] V. Bartsch, R. Machado, D. Merten, M. Rahn and F.-J. Pfreundt, "GASPI/GPI In-memory Checkpointing Library", in *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings* (F. F. Rivera, T. F. Pena and J. C. Cabaleiro, eds.), Lecture Notes in Computer Science, Springer, 2017, pp. 497–508.

[63] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming", *IEEE Comput. Sci. Eng.* **5** (1998), no. 1, pp. 46–55.

[64] O. Krzikalla, A. Rempke, A. Bleh, M. Wagner and T. Gerhold, “Spliss: A Sparse Linear System Solver for Transparent Integration of Emerging HPC Technologies into CFD Solvers and Applications”, in *New Results in Numerical and Experimental Fluid Mechanics XIII* (A. Dillmann, G. Heller, E. Krämer and C. Wagner, eds.), Springer, 2021, pp. 635–645.

[65] A. Griewank, D. Juedes and J. Utke, “Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++”, *ACM Trans. Math. Softw.* **22** (1996), no. 2, pp. 131–167.

[66] H. Jasak, “OpenFOAM: Open source CFD in research and industry”, *Int. J. Nav. Archit. Ocean Eng.* **1** (2009), no. 2, pp. 89–94.

[67] D. Molinero, S. Galván, J. Pacheco and N. Herrera, “Multi GPU Implementation to Accelerate the CFD Simulation of a 3D Turbo-Machinery Benchmark Using the RapidCFD Library”, in *Supercomputing* (M. Torres and J. Klapp, eds.), Springer, 2019, pp. 173–187.

[68] P. He, C. A. Mader, J. R. Martins and K. J. Maki, “Dafoam: An open-source adjoint framework for multidisciplinary design optimization with openfoam”, *AIAA J.* **58** (2020), no. 3, pp. 1304–1319.

[69] M. Sagebaum and N. R. Gauger, *MeDiPack — Message Differentiation Package*, 2024. Online at https://scicomp.rptu.de/software/medi/ (accessed on October 31, 2025).

[70] F. Palacios, M. Colonno, A. Aranake, et al., “Stanford University Unstructured (SU^2): An open-source integrated computational environment for multi-physics simulation and design”, in *51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition 2013*, American Institute of Aeronautics and Astronautics, 2013, (60 pages).

[71] N. Beishuizen and U. Agrawal, *SU2GUI*, 2024. Online at https://github.com/su2code/su2gui (accessed on October 31, 2025).

[72] P. Singh and A. Manure, “Introduction to TensorFlow 2.0”, in *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*, Apress, 2020, pp. 1–24.

[73] S. Imambi, K. B. Prakash and G. R. Kanagachidambaresan, “PyTorch”, in *Programming with TensorFlow: Solution for Edge Computing Applications* (K. B. Prakash and G. R. Kanagachidambaresan, eds.), Springer, 2021, pp. 87–104.

[74] H. Topcuoglu, S. Hariri and M.-Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing”, *IEEE Trans. Parallel Distrib. Syst.* **13** (2002), no. 3, pp. 260–274.

[75] L. F. Bittencourt, R. Sakellariou and E. R. M. Madeira, “DAG scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm”, in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, IEEE, 2010, pp. 27–34.

[76] H. Arabnejad and J. G. Barbosa, “List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table”, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014), no. 3, pp. 682–694.