Towards a Modular and Variability-Aware Aerodynamic Simulator

Towards a Modular and Variability-aware Aerodynamic Simulator *

Ferruccio Damiani[1], Michael Lienhardt[2], Bruno Maugars[2], and Bertrand Michel[2]

[1] University of Turin, Turin, Italy
ferruccio.damiani@unito.it
[2] ONERA, Palaiseau, France
{michael.lienhardt, bruno.maugars, bertrand.michel}@onera.fr

Abstract. Computational Fluid Dynamics (CFD) consists of numerically solving the fluid dynamics equations and has become a major tool in designing and evaluating any physical structures, like airplane, rotors, or even nuclear plants, where the flow of a fluid can be a critical efficiency or security aspect of these structures. Our first contribution is a brief review of the core characteristics a CFD solver should have (based on two common functionalities they usually provide) and the state of the art of CFD tools. Indeed, research on this field principally focuses on specific numerical or computation methods, software architecture is rarely discussed. Moreover, to the best of our knowledge, all CFD tools have major structural flaws that limit their capacities to integrate new methods and take advantage of new hardware. Our second contribution is a new approach that aims to solve these flaws. We exploit formal methods (namely, *order-sorted algebra* and *Delta-Oriented Programming*) to build a flexible CFD framework in which new methods can be added as modules. By exploiting *dataflow automatic generation*, our approach adds no runtime overhead. We implemented our approach and tested it on a simple example.

1 Introduction

Over the past 30 years, aerodynamic numerical simulation tools (also called CFD tools) [41, 28, 48, 8] has been largely used and become essential for the development, sizing and maintenance of products manufactured in the aeronautics sector, like airplanes, turbines, etc. These tools calculate the flow properties and the mechanical stress (like a wind shock, a drag or a lift) applied on the manufactured products, and this information is used by the designers of the different products to guide them in their tasks (e.g., development, sizing or maintenance). Aerodynamics is described by the Navier-Stokes (NS) equations [12] which have no known analytic solution [38], and so, CFD tools are structured around two

* The authors of this paper are listed in alphabetical order. This work was partially supported by the SONICE project, granted by the French Directorate General for Civil Aviation (DGAC).

approximations: first, they use equations that approximate NS; second, they use physical configurations that approximate the volume in which the fluid flows. Many equations and many meshes have been designed over the years, each of them having a specific usage: some are more suited to some specific physical conditions (e.g., supersonic speed), some trade-off precision for efficiency. But even for very efficient equations and meshes, on realistic usecases these computations are very memory and computation intensive and require massive and efficient hardware.

Consequently, CFD tools face two design challenges that seem in opposition. On one hand, they must be flexible enough to support a large catalogue of NS approximations and meshes, so they can be used to analyse different manufactured products. Moreover, since the research on NS approximation and on meshes is very active, the CFD tools must regularly be updated to integrate these new results, which also requires flexibility. On the other hand, these tools must be very specific and close to the hardware in order to be as efficient as possible: a simple cache-miss in a loop could have desastrous effect on the computation time and make the tool useless in practice. Moreover, *heterogeneous computations* (i.e., distributing the computation on different kind of computation units, like CPU and GPU) must also be fine-tuned, to avoid costly transfert of control and data between the different computation units. Finally, even though CFD tools must be fine-tuned to take as much advantage of the hardware as possible, the workstations on which these tools run are all different and in constant evolution, with the rise of new hardware technologies (GPU, TPU, VE [19], etc).

In this paper, on the occasion of Reiner Hähnle's 60^{th} birthday, we present our ongoing research on an approach for solving the apparent incompatibility between the requirements of flexibility and fine-tuning of a CFD tool. This approach is based on *Delta-oriented Programming* [53, 26, 14, 15] and on an ad-hoc code generation to enable flexibility and fine-tuning, respectively. Reiner Hähnle, as part his academic work, provided outstanding contributions in the development of formal methods and tools for supporting rigorous software engineering approaches – see, e.g., the KeY tool [2] and the ABS modelling language [24]. Notably, his research has always looked at practical applications – see, e.g., the EU FP7 project HATS (Highly Adaptable and Trustworthy Software using Formal Models) [22], the EU FP7 coordinating action Eternals (Trustworthy Eternal Systems via Evolving Software, Data and Knowledge) [23], the EU FP7 project Envisage (Engineering Virtualized Services) [25], and the DB Netz AG project FormbaR (Formal modelling and analysis of Railroad operations) [33]. We therefore believe that the research activity reported in this work fully falls in Reiner Hahnle's research interests.

Outline. Section 2 briefly outlines Computational Fluid Dynamics (CFD) and its challenges. Section 3 describes the characteristics of the current approaches, focusing in particular on the elsA tool [8]. Section 4 introduces our approach, and Sections 5 and 6 focus on the data model and the variable operators aspects of the approach, respectively. Section 7 present our initial results. Finally, Section 8 concludes the paper and provides and outlook on future work.

2 Computational Fluid Dynamics Challenges

This Section presents the different characteristics required of a CFD tool, and the challenges in implementing them. We structure this Section in two parts: first, we discuss the *functional characteristics* of such a tool, i.e., what are the functionalities expected by the user; second, we discuss its *computational characteristics*, i.e., how to perform the actual computation on a given hardware.

2.1 Functional Characteristics

Equilibrium State Computation. The main functionality of a CFD tool consists of computing an *equilibrium state* [50] of a given physical configuration [52, 51]. This state usually consists of the temperature, the velocity and the pressure of the fluid, at every point of the volume considered in the given physical configuration.

Consider for instance the configuration depicted in Figure 1. This configu-

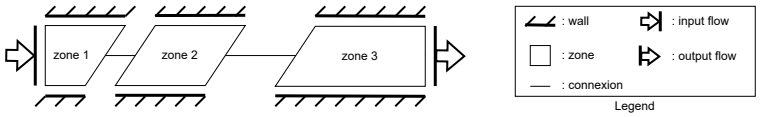

Fig. 1: CFD Physical Configuration Example

ration corresponds to a simple 2D tube: the space where the fluid can flow is modelled by the three connected zones `zone 1`, `zone 2` and `zone 3` bounded by walls on the top and on the bottom, with an input gas flow on the left and a possible exit flow on the right. Depending on the fluid property, the input and output flow conditions, the friction on the walls, the computation could return an equilibrium state that can greatly vary.

The principle of the equilibrium computation is quite simple: the different zones in the configuration are implemented with meshes storing default values (2D meshes in our example in Figure 1), and the constraints given by the different *boundary conditions* (BC) in the configuration (the input flows, the output flows and the wall in our example) are iteratively propagated in the meshes, thus changing the stored values until the value modifications derived from these constraints are negligible.

Numerical Optimization. Another essential functionality integrated in most CFD tool is called *Numerical Optimization*. Any realistic simulation depends on many parameters, e.g., the shape of a plane wing, that can have a great influence on some *objective functions* that should be minimized, e.g., the plane drag. A first approach to minimize these objective functions is to perform a large number of simulations, each of them with a different parameter configuration. This approach is not very practical however, due to size of the configuration space to explore.

An interesting alternative approach is to compute the derivative of the objective functions: since the zeros of these derivatives correspond to local minima and maxima of the objective functions, it is enough to perform the computation on these configurations, thus greatly reducing the search space. There exist two main approaches to compute these derivatives. The first one introduces a perturbation to the problem inputs and computes theirs influences on the result of the objective functions (this approach is called *forward* or *linearised* mode). It is also possible to introduce a perturbation to the result of the objective functions and study their influences on the inputs (this approach is called *backard* or *adjoint* mode). The two modes of computing this derivate have different properties: the forward/linearized mode is more efficient when number of objective functions is greater than the number of parameters, while the backward/adjoint mode is more efficient in the opposite case. In practice, the number of parameters is several order of magnitude bigger that the number of objective functions, and so the backward/adjoint mode is the most efficient. However, the backward/adjoint mode add huge constraints in term of software architecture because all the derivatives must be propagated and computed backwards [34].

Kenway et al. in [34] give an in-depth discussion on the different methods to compute the derivatives necessary to solve the numerical optimization problem, and give clear motivations why *discrete adjoint* approaches must in general be preferred over *continuous direct* approaches. Additionally, they compare different implementations for computing such a discrete adjoint, and a code generation technique called *Automatic differentiation* (AD) gives the best result, in term of memory usage, speed and accuracy.

AD thus is a very powerful technique, but it has one important limitation. It is a source to source transformation techniques that generates from the implementation of a function a code computing the derivative of that implementation. And in order to produce correct code, AD expects that the input implementation follows a simple workflow pattern.

Hence it is important to structure a CFD tool in a way so that the computation it performed is expressed as a simple workflow that matches the AD restrictions.

Functional Variability. While the computation of a physical configuration's equilibrium state is always a fixpoint loop, one of the main difficulty in designing a CFD solver is to manage the fact that the content of that loop has a very large number of variants, and that this number is always increasing. This very large variability has three causes, two of which we already introduced:

1. **The approximation method.** As previously discussed, many approximation methods for NS have been and are still being designed [4, 36, 29, 32], each of them having their own advantages and disadvantages, e.g., are more suited to specific physical configurations, to specific data computation, etc. Additionally, many of these methods use constants (modelling some physical properties) that must be set by the user. Finally, some of these methods are designed in a way so they are incompatible with other variable elements, e.g., some physical configurations.

2. **The physical configuration.** Each configuration is unique and requires a tailored computation. First as previously discussed, the BCs describe the constraints on the flow of the fluid going through the space modelled by the zones, and each of them has a specific implementation. Then, the flow follows the links between the zone, and so the topology of the physical configuration has a direct impact on the computation.
   Of course it is possible to design (as it has already been done in the past) a unique spaghetti code that can manage all possible physical configurations. However such a code would be unmaintainable and highly inefficient. Indeed, such a centralized code needs to have direct access to all the arrays and matrices during the physical simulation, which causes important latency in accessing the memory for physical configuration of regular size. This issue is discussed in more detail in Section 2.2.
3. **The user requested data.** The user can request the computation of some specific data (e.g., an objective function), which must be included in the fixpoint loop. The computation of this data may require internally the computation of some other temporary information on the flow of the fluid, which adds a layer of complexity in the construction of the fixpoint loop. Moreover, in some case the precision of the requested data can be configured, which in turns may require to change how the temporary information are computed.

Runtime Checkup. Finally, it could be very useful to be able to insert monitoring and controlling capabilities at key points in a CFD computation. Indeed, such computations can take a very long time. So, it could first be very useful to be able to regularly store a snapshot of the current computation so in case of *hardware failure* we might not lose hours or even days of computation. Moreover, *convergence* of the fixpoint loop is not guaranteed in many cases: monitoring and controlling capabilities could be very useful to detect when the computation is not converging and to update some of the solver's options in order to solve the problem, or stop the computation if no solution can be found.

2.2 Computational Challenges

Like many other HPC applications, CFD is in general very memory and computation intensive, and so it is very important to use as efficiently as possible the available hardware.

Distribution. The first difficulty in using efficiently the hardware is data locality [58, 35, 46]. Indeed, in many cases the meshes of a physical configuration count several millions or even billions of points, and standard *SMP* memories (that can be accessed uniformly by all the CPU in a workstation) cannot scale to such sizes: the latency in accessing the memory becomes too big. The *NUMA* memory design (which stands for *Non Uniform Memory Access*) solves this scaling issue by structuring the memory in *nodes*, each one having a guaranteed good latency with one CPU. Consequently, it is important to partition the meshes in

chunks that can be stored in one node of memory, and to partition the computation so that each part of the computation is performed on the CPU close to the data it manipulates.

Communication. Some functions, called *stencil functions* [30, 9], compute a value on a node of a mesh by looking at the neighbours of that node (similarly to the *convolution* operation in Artificial Intelligence). However, since the mesh is distributed, some neighbours are not locally available on a CPU: it can be necessary before running a stencil function to fetch the value of these neighbours from the NUMA node that hosts them.

Computation Reordering. A well known method to optimize catch access is to reorder some computation, so that those that use the same data are executed together. This optimization is implemented in most compilers, but can only be applied on a fine-tuned program: any CFD tool with some flexibility will not be optimized by the compiler. Hence it is important to find a way for a CFD tool to perform this optimization itself.

Heterogeneous Hardware. Now-a-days, workstations have several kind of processing units (PU), e.g., CPU and GPU. Moreover, several means of communication (with different properties) exist between these PUs, e.g., PCIe and NVLink. Since some computation are more efficient on some hardware (e.g., a GPU handles well repetitive computation over large sets of data), and some cannot be performed on them (e.g., GPUs do not have function pointers), it is important to design a distribution plan that put the computation on suited PUs, while taking in account the latency of data transfer.

Variable Hardware. The final difficulty is to be able to manage the fact that a CFD tool will be executed on several workstation, each of them with its own hardware. Hence, the hardware itself becomes variable in this context, and the distribution plan discussed in the previous paragraph must be generated and tailored for workstation running the tool.

Figure 2 illustrates the shape of the a possible distribution plan of the physical configuration of Figure 1. In this example, we consider an hardware architecture with two NUMA nodes, the first one hosting `zone 1` and `zone 2` and the second one hosting `zone 3` of our physical configuration example of Figure 1. The first NUMA node is linked to a dual core, the first core having two threads while the second having only one. The second NUMA node is linked to a highly parallel architecture, like a GPU. The arrows between NUMA nodes and caches represent the different communications that are necessary for the computation of the equilibrium states.

Communication between a NUMA node and the local caches is quick and all the data stored in the node must at one point of the computation be sent in the cache. But it is still better to avoid useless transfer between the NUMA node and the cache. Communication between NUMA nodes is slower, and should be

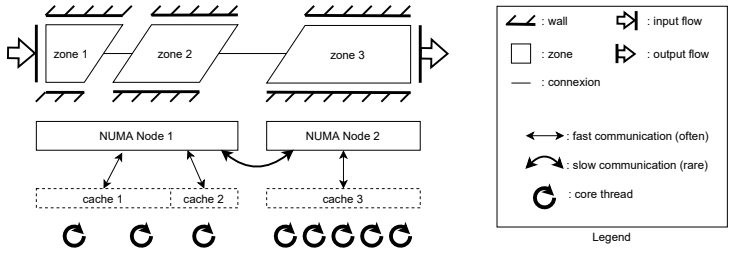

Fig. 2: CFD intended Running Architecture

perform only when necessary. In opposite to the communication between NUMA nodes and caches however, the data exchanged between NUMA nodes is far less than the whole content of the nodes.

3 State of the Art

In this Section, we give a brief presentation of the main CFD tools, and then focus on elsA [8] which has been developed at ONERA for the last 20 years. Many mature tools (like elsA) have been developed in research centre but in close relation to industry. Consequently, while documentations on how to use these tools are freely available, in-depth description of their internals and how they deal with the challenges presented in Section 2 are (to the best of our knowledge) not published. This is the case of FUN3D [5] (developed at NASA), TRACE [48] and Flucs [39] (developed at DLR), and elsA [8] (developed at ONERA).

On the other hand, less mature but more documented open source CFD tools are now available. SU^2 [49] and OpenFOAM [31] are developed in pure C++, and largely advertise the use of classes and inheritance to: *i*) structure their code in modules; *ii*) reuse the code in different part of their toolchain; and *iii*) use uniform APIs to have more generic code. However, having modular and generic code is not enough to capture all the flexibility expected from a CFD tool, and all the management of the user requested data, of the input physical configuration and of the hardware must be implemented directly by the user. Moreover, these tools do not implement numerical optimization and can only run on CPUs (due to the language limitation).

pyFR [57] is a tool implemented in python and uses the many libraries available in this language to perform quite well. While the `sympy` library is used to provide an abstract DSL in which the user can write his mathematical formula, the orchestration of these formula (how and when they are executed) must be written in python by the user. Then, at runtime, when a mathematical formula must be executed, pyFR translates it into C or CUDA code (for an execution on CPU or on GPU), and compiles and runs the code. This tool does not implement numerical optimization. Devito [40] is similar to pyFR. It also uses the

`sympy` library to express mathematical formula in python, but uses its own DSL to orchestrate them. That way, the whole computation (the formula and the orchestration) can be translated into C code and run in parallel. However, since GPUs cannot perform orchestration, Devito does not run on GPU for now. While pyFR and Devito have very interesting approaches, these tools suffer from the same main issues as SU^2 and OpenFOAM: the orchestration of the mathematical formula (which includes the management of the user requested data, of the input physical configuration and of the hardware) must be implemented directly by the user.

3.1 elsA's Approach

elsA is a mature CFD tool and solves, at least partially, the gap between flexibility and fine-tuning. Its solution relies on its 3 parts structure which, for historical and performance reasons, are all implemented in a different language. First, similarly to pyFR and Devito, elsA distinguishes between mathematical functions and orchestration: mathematical functions are called *operators* in elsA and are implemented in Fortran 90 [45][3] which is a very efficient language for physic simulation and that is simple enough to support automatic differentiation; orchestration is implemented in python, but in the opposite of pyFR and Devito, it is handled by elsA directly and requires almost no setup by the user. The third part of elsA, called *HPC layer*, handles the hardware, in particular the management of the distribution of the computation, and is implemented in C++17 [55].

The transition between flexibility and fine-tuning is handled by an initial analysis of all the inputs by the orchestrator, which produces a plan of which function to execute, in what order and on which hardware. This initial analysis is structured in 3 steps, and once it completes, the actual computation, i.e., the execution of the generated plan, starts.

Step 1: Loading the inputs. First, elsA loads the different inputs:

- It queries the available MPI [47] library for the NUMA and CPU structure (GPU are not supported by elsA). elsA at this stage assumes for simplicity that the only kind of computational units in the hardware architecture are identical CPUs. This hypothesis ensures that the cache sizes of all CPUs are the same.
- The physical configuration is loaded from a CGNS file [52, 51], which is a standard format for CFD physical configuration. This file format stores among other data the topological structure of the physical configuration with all the BC setups, which makes it de facto one of the main standards to store this part of the configuration space.

[3] Mathematical DSLs like `sympy` that could be translated in efficient code did not yet exist when elsA was already mature.

– The user requested data is also loaded from the CGNS file. They are specified in special entries, distinct from the ones describing the topology of the physical configuration. These entries inform elsA about which data to compute and on which zone to compute it. Currently, the set of data the user can request is fixed (this set is specified by an enum in the elsA API).
– The approximation method options are loaded from a elsA-specific python file. There is no well structured and clear management of the options in elsA. In order to manage the very large number of option, an initial dependency mechanism has been implemented, based on an ad-hoc usage of python dictionaries. But this system is difficult to maintain and cannot express all the dependencies and conflicts the options actually have, and currently, it only performs basic checks while many configuration errors are detected during computation.

Step 2: Managing Hardware Flexibility. Once the setup has been loaded, elsA uses its homogeneous hardware hypothesis to uniformly distribute the physical configuration over the CPUs. This is done by splitting the zones into subzones and distributing them over the NUMA nodes, so that every subzones are hosted on one unique NUMA node, and that all this data is equally distributed between the available CPUs. Additionally, elsA reorders the information within each subzone so that all data that should be access together are contiguous in memory, thus avoiding useless cache misses.

Note that information about the structure of the data distribution is kept by elsA's HPC layer which is responsible of managing the distribution of the computation. That way, it is able to give in parameter to each executing operator the data hosted in the local NUMA node.

Step 3: Managing Functional Flexibility. In order to manage the approximation method options and the user requested data, elsA uses an ad-hoc and powerful architecture that generates the list of all the operator to execute for each CPU on the workstation. This architecture, called *Factory*, is illustrated in Figure 3.

The factory is structured in two parts. The first part is a hard-coded registration of all the operators available in elsA with: *i*) their dependencies (i.e., when some input data is computed by another operator); and *ii*) how they are triggered or disabled by the different inputs. The dependencies and triggers are mainly implemented with simple `if` conditions and result in a rather large spaghetti code. This part takes in input the approximation method options and the user requested data, and produces an initial, non optimized list of operators to execute. The second part of the factory cleans and restructure the list of operator to make it more efficient by applying standard optimization techniques, like operator reordering. Moreover, this part also insert communication operators in the list to ensure that the local data is consistent with the data on other CPUs.

The result of the Factory's computation (on the right of Figure 3) is a list of operators and communications to be executed for a given subzone: the factory is executed on every CPU hosting data, to compute what this CPU needs

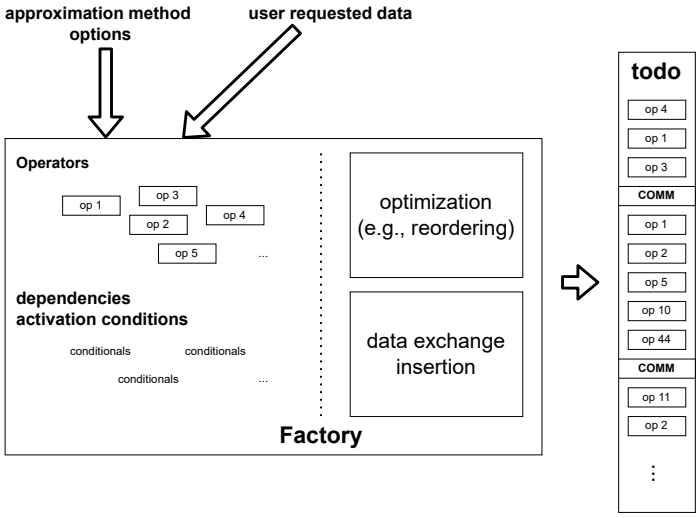

Fig. 3: elsA Factory

to do. Moreover, this list has an essential property for the numerical optimization capabilities of elsA: it can be efficiently differentiated. Indeed, since all the operators (implemented in Fortran) can be automatically differentiated, so is a simple sequence of these operators.

3.2 elsA's Approach Limitations

While elsA and its workflow is used in production to solve complex industrial usecases, it should be clear now that it has strong limitations both in its management of the hardware and functional flexibility.

Hardware Flexibility. The first limitation of elsA is its uniform CPU architecture hypothesis. Heterogeneous architectures involving different kind of computational units are getting ubiquitous [1, 18], and this hypothesis is simply no longer realistic.

Functional Flexibility. In this context, elsA suffers from 4 main issues. The first one concerns the approximation method options. elsA currently contains more that 2000 of these interdependent options, without any validation tool: when the user is lucky, an erroneous configuration makes the factory fail and gets an error message almost instantaneously; however for many erroneous configurations, the factory can generate a plan, and the user needs to wait the result of the computation to see that something went wrong, without knowing where.

The second one is the difficulty to maintain the specification of the dependencies between operators and their activation conditions. Simple conditionals do not scale to manage hundreds of operators and thousands of options.

The third one concerns numerical optimization. While the operator list generated by the factory can be differentiated, many operators and communications in that list are not relevant for the derivation of many objective functions, and so the automatic differentiation implemented in elsA performs many useless computation and should be optimized.

Finally, elsA is too restrictive in its specification of user requested data. Indeed, as previously stated, elsA only provides a fixed list of possible data to compute to the user. This limitation make it so that every time a user wants to analyse some new data, or some new interesting objective functions is designed, the user needs to ask the elsA team to implement the computation of that specific data, even if all the operators necessary to compute it are already implemented. This could cost a lot time and effort to the user and the elsA team, and having a more generic approach to user request could significantly reduce this cost.

4 Our Approach

As stated in the introduction, the goal of our approach is to bridge the gap between the requirements of flexibility and fine-tuning in a CFD tool. This step is necessary to answer the different challenges described in Section 2. To achieve this, our approach follows the 3 parts structure of elsA, with a complete redesign of the factory. Indeed, structuring the computation in elementary operators is necessary to manage flexible hardware without cluttering the computation code with concurrency concerns; the factory's automatic generation of a plan is necessary to be able to seamlessly use the tool with different configurations; and the HPC layer is necessary for the management of the actual computation.

Hence, the main novelty of our approach is a new factory, whose architecture is presented in Figure 4. This architecture is structured in 4 parts: one for the *graph generator* and one for each of the factory's input.

The Graph Generator. The core idea of our approach is to replace the spaghetti code in the factory with a clear notion of dependencies between operators. That way, the generation of the plan simply corresponds to a dependency resolution, which results in a *Directed Acyclic Graph* of operators instead of a list. Incidentally, this graph is actually exactly what is needed to avoid the problem elsA has with numerical optimization: since the dependencies between operators are explicit, we can identify the operators that are necessary for the computation of a specific objective function, and only derivate them.

We implement the notion of dependency by requiring the developer to specify the semantics of the inputs and outputs of each operators. Indeed, while the actual inputs and outputs of the operators are usually arrays of double, the semantics of the contained values, e.g., the fluid density or the gradient of the temperature, is specific to each operator. This specification is similar to type annotations, except that an operator can have several outputs. Moreover, some operators require special care, like the gradient operator because it can compute the gradient of any value. its specification corresponds to a type of the form $\alpha \rightarrow$

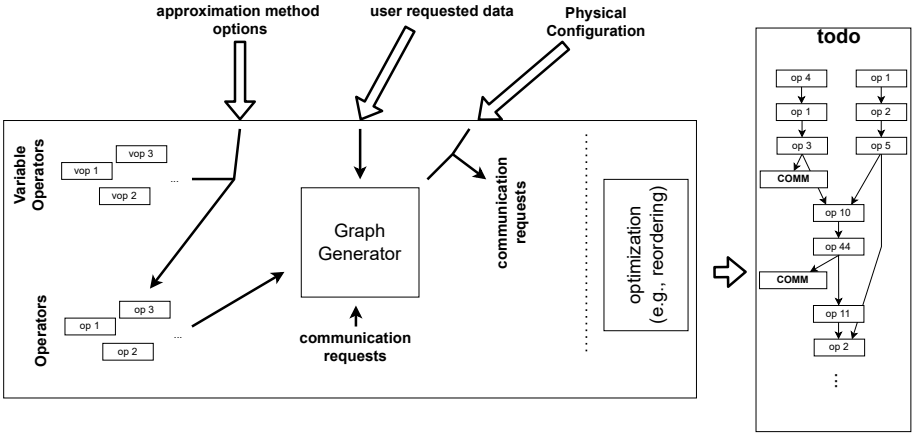

Fig. 4: New Variability Management

`grad`(α). For simplicity and genericity, we thus use *terms* for our specifications, and will give more details on our usage of terms and our implementation in Section 5.

Managing the Options and the Operators. Selecting or not options changes the implementation of related operators. Moreover, depending on which implementation is used, the inputs and outputs of the operator may vary.

To deal with this variability, we use *Software Product Lines* (SPLs) [56, 13, 3, 53], and more precisely *Delta-Oriented Programming* (DOP) [53, 14, 15] to make the specification of the operators variable w.r.t. the options selected by the user. We will detail this part in Section 6, but for now it is enough to know that applying a specific set P of selected options on a *variable operator specification* returns the specification of this operator's implementation for P.

Managing the User Requested Data. Our approach uses the same terms for the User Requested Data and for the operator specifications. That way, such a request can be considered like any other dependency by our graph generator.

Managing the Physical Configuration. The physical configuration is used by our graph generator to identify the communications that need to be inserted in the graph, and where. Indeed, during computation, stencil operators require fetching specific values from the neighbours of the zone on which the stencil is being executed. To illustrate how this requirement is managed in our approach, let first state that similarly to elsA, we have one graph generator per PU, that generates the graph to execute on that PU. Let now consider a specific PU, and note G the graph generator for that PU, and if Z is a zone, then $neigh(Z)$ is the set of all neighbours of Z. Upon inserting a stencil operator working on a zone Z in the graph, G also adds the corresponding `receive` operators with all the

zones in *neigh*(Z) and sends a *communication request* to the graph generators managing the zones in *neigh*(Z). Upon the reception of a communication request, G adds a corresponding `send` operator in the graph: that operator depends on the requested data, and so the dependency analysis will ensure that this data is computed before being sent.

The two following Sections will go more in-depth into two aspects of this new variability management: Section 5 discusses the usage and implementation of terms in our approach; and Section 6 details the notions of SPLs and DOP, and discusses the implementation of the variable operators.

5 Data Model

As described in Section 4, we use terms to specify the inputs and outputs of our operators. Additionally, we use *order-sorted Algebra* [44, 16] to specify which terms are valid inputs and outputs. This combination is particularly suited to our needs: terms offer a very flexible structure to specify the data exchanged between operators, and such flexibility is necessary when considering the maintainability and future evolutions of the tool; on the other hand, order-sorted Algebra is used to ensure that the user gives at least a sensible specification to his operators[4].

Finally, terms support efficient *pattern matching* (a subcase of term unification [17] where one term is ground), which is a functionality required by the graph generator: solving a dependency corresponds to finding an operators that has one output matching that dependency.

5.1 Implementation

While order-sorted algebra has been implemented in various formal specification tools [21, 11], to the best of our knowledge no existing implementation can be used in our approach. Indeed, our approach requires the order-sorted algebra implementation to provide the following three elements:

- a simple syntax to specify the inputs and outputs of an operator;
- a pattern matching API that can be used by an external graph generator;
- a mean to integrate the syntax in external transformation function used to generate operator specifications.

Consequently, we implemented our own library, and choose python for the implementation language. Python is a popular language in which to embed Domain Specific Languages (DSLs) due to its very flexible syntax and its readiness to support C and C++ libraries [59, 6, 27]. Moreover, embedding a DSL in an existing language allows for its seamless integration with other functionalities

[4] The flexibility of terms and ease to specify algebra was also a key element in the development of our approach: many trials and errors went into the design of a term structure that captures the necessary features of a CFD data.

available in the language. That way, all three requirements we listed are satisfied: we have a DSL for the syntax requirement; and its integration in python answers to last two requirements. In particular, it allows for the integration of this library with our other library implementing Delta-Oriented Programming and presented in Section 6.

First we designed the following DSL to specify a signature:

$sig ::= \texttt{declare\_sig}(\overline{srt}, \texttt{order} = (\overline{od}))$	Signature Declaration
$srt ::= id = (\overline{ct})$	Sort Declaration
$ct ::= (id, \overline{id})$	Constructor Declaration
$od ::= (id, id)$	Order Declaration

As usual, $\overline{X}$ denotes a possibly empty finite sequence of elements X and $[X]$ denotes that the element X is optional. This DSL does not follow standard signature declaration like in Maude [11] because of the limitation of Python syntax. A signature is declared using the `declare_sig` function, and is composed by the declaration of a list of sort, plus some partial ordering between sorts. A sort declaration *srt* first gives a name *id* to the sort, and introduces the set of constructors of this sort. A constructor declaration *ct* is a tuple of names *id* where the first one is the name of the constructor, and the others are the sorts of the different parameters of the constructor. Finally, an order declaration *od* simply gives a order relation between two sorts.

Example 1. A simple signature for natural numbers can be described as follow:

```
sig_nat = declare_sig(
 nat = (
  ("zero",),
  ("succ", "nat")
))
```

Here, the signature `sig_nat` contains one sort called `nat` and two constructors: `zero` of sort `nat`, with no parameter; and `succ` of sort `nat`, with one parameter of sort `nat`.

Once a signature has been defined, it is first possible to extend it by calling: the method `add_sort(`*id*`)` which adds a new sort named *id* to the signature; or the method `add_constructor(`*id*`,` *id*`,` $\overline{id}$`)` where the first parameter is the name of the constructor's sort, the second parameter is the name of the constructor, and the other parameters are the names of the sort of the constructor's parameters.

It is also possible to create terms. The method `fresh_variable(`*id*`)` returns a fresh variable of sort *id*. Structured term construction uses the Python lookup API to make term constructors directly available as fields or methods of a signature. For instance in the context of Example 1, the expression `sig_nat.succ(sig_nat.zero)` corresponds to 1.

Finally, pattern matching is available with the method `match`. This method returns `None` if the pattern matching fails, or a substitution that can be applied on a term.

Example 2. To illustrate the term construction and pattern matching of our library, let consider the following python code:

```
1 plus_one = sig_nat.succ(sig_nat.fresh_variable("nat"))
2 two = sig_nat.succ(sig_nat.succ(sig_nat.zero))
3 subst = sig_nat.match(plus_one, two)
4 if(subst is not None):
5  assert(subst(plus_one) == two)
```

Line 1 creates a nat term called `plus_one` containing a fresh variable. Line 2 creates a term called `two` corresponding to the number 2. Line 3 matches `plus_one` against `two` and stores the result in `subst`. By construction, the pattern matching succeeds, and the result is the substitution mapping the variable to `sig_nat.succ (sig_nat.zero)` (which corresponds to the number 1). In line 5 we check that the computed substitution is correct, by ensuring that applying it on `plus_one` does return the term `two`.

5.2 Application to CFD

In a simple setting, CFD data can be specified with a triplet. The first component corresponds to the a *value* stored in the data, like `Density` or `grad(Momentum)`. The second is a location, i.e., on which element of a mesh that value is placed; possible locations on a 3D mesh are `vertex`, `edge`, `face` or `cell`. The third component is the id of the zone (i.e., the mesh) where the data lives.

The following code excerpt presents a part of the signature we designed:

```
1 cfd_sig = declare_sig(
2  data = ( ("data", "value", "location", "zone_id"), ),
3  value = (
4   ("Density",), ("Energy",), ("Momentum",),
5   ("grad", "value"),
6  ),
7  location = ( ("cell",), ("face",), ("edge",), ("vertex",) ),
8  zone_id = ( ("zero",), ("succ", "zone_id") ),
9 )
```

We model an data with the `data` constructor (of sort `data`), declared in Line 2. This data takes three parameters, respectively of sort `value`, `location` and `zone_id` . A value can either be base values like `Density` `Energy` or `Momentum`, or structured ones like the `grad` of another value. As previously discussed, we have four constructors for locations, and `zone_id` are modelled like natural numbers.

The following code excerpt illustrates our signature by specifying the input and output data of the gradient operator.

```
1 vzone = cfd_sig.fresh_variable("zone_id")
2 vvalue = cfd_sig.fresh_variable("value")
3
4 gradient_input = cfd_sig.data(vvalue, cfd_sig.cell, vzone)
5 gradient_output = cfd_sig.data(
6   cfd_sig.grad(vvalue), cfd_sig.cell, vzone)
```

We first declare two variables, one for the zone of the input data of the operator, and one for its input value. Then line 4 states that any data whose location is `cell` is a valid input to the gradient operator. Line 6 on the other hand states that the output data of the gradient operator is also on `cell`, on the same zone as the input data, and its value is the `grad` of the input value.

6 Variable Operators

As described in Section 4, we use SPL [56, 13, 3, 53] and DOP [53, 14, 15] to manage both: the relationship between the approximation method options; and how the operators' implementation and specification are affected by the selection of these options.

SPL corresponds to the concept of managing a collection of similar software artefacts that are characterized by the set of *features* they implements. The selection of a set of feature is called a *product*, and the artefact corresponding to that product is called *the product's variant*. One key aspect of an SPL is the explicit specifications of its features' dependencies and incompatibilities. For instance, firefox can be compiled with the gtk or aqua graphics library, but not both at the same time: these two features are incompatible. *Feature Models* [56, 13] are a standard way to declare the relationship between features.

DOP is a *transformative* approach to implement SPLs, i.e., a product's variant can be obtained by applying the set of transformations (called *delta*) corresponding to the product on a initial artefact. DOP structures an SPL in 4 parts: a *feature model* gives the features of the SPL and their relationship; an *initial artefact* gives the starting point for the generation of all variants of the SPL; a global *set of deltas* lists all the transformations that can be applied during the computation of a product's variant; and *configuration knowledge* maps every delta to the set of products that activate its execution, and also states in which order delta must be applied. The activation set of a delta is usually specified with a Boolean formula over the features of the SPL.

In our approach, we wrap every operator specification in a DOP product line, where deltas can: add new inputs and outputs to the initial specification; and change the link to the actual implementation of the operator. Moreover, all these SPL share a common Feature Model that lists all the available options of the CFD tool and their relationship. That way, we have a clear way to ensure that the options selected by the user are correct or issue a message stating which relationship is being broken before any computation happens.

While DOP has been implemented for several types of artefacts [37, 10, 54], to the best of our knowledge, no existing implementation can be used in our approach. Indeed, while the framework presented in [54] is generic enough to express DOP product lines over operator specifications, it has two major drawbacks: *i*) the amount of implementation to use this framework is disproportionate compared to the simple structure of an operator specification; and *ii*) it only considers product lines in isolation and thus cannot share a common feature model between different SPLs.

6.1 Implementation

Our implementation of DOP follows the same principles of our implementation of terms and is structured in two DSLs: one for the Feature Model and one for the definition of DOP product lines.

First, we designed the following Feature Model DSL, based on existing representation [56, 13]:

fd	$::= \texttt{FD}(id, \overline{fatt}, \overline{fg}, [ctc])$	Feature Diagram
fg	$::= fop(\overline{fd})$	Feature Group
fop	$::= \texttt{FDAnd} \mid \texttt{FDAny} \mid \texttt{FDOr} \mid \texttt{FDXor} \mid \ldots$	Feature Group Operations
$fatt$	$::= \texttt{Att}(id, \text{domain})$	Feature Attribute
ctc	$::= id \mid \text{Pred}(\overline{id}) \mid \texttt{And}(\overline{ctc}) \mid \ldots$	Cross Tree Constraint

As usual, $\overline{X}$ denotes a possibly empty finite sequence of elements X and $[X]$ denotes that the element X is optional. This DSL fits Python syntax and describes a feature diagram with attributes and cross-tree constraints. A Feature diagram fd declares a feature id with possible associated attributes $\overline{fatt}$, can have sub-trees identified by a set of feature diagram groups $\overline{fg}$ and may have a cross-tree constraint ctc linking features and attributes declared in its sub-trees. A Feature diagram group fg gives a constraints fop on a set of feature diagrams $\overline{fd}$: `FDAnd` means that all diagrams must be selected; `FDAny` means that all diagrams are optional; `FDOr` means that at least one diagram must be selected; and `FDXor` means that exactly one diagram must be selected. Attributes $fatt$ have a name id and a domain, which is left unspecified in this grammar (in elsA, it is expected that most of these attributes would be floats or arrays of floats). Finally, cross-tree constraints ctc are generic SAT constraints over feature names id, extended with domain specific predicates (e.g., float comparison).

Second, we implemented a very simple API to declare product lines and add deltas to it. The following line declares a new product line named `spl`, with configuration space `fm` (e.g., a feature diagram as discussed previously) and core product `core`:

```
spl = SPL(fm, core)
```

Declaring a delta to the product line `spl` is done as follows:

```
@spl.delta(ac)
def delta(variant, product):
 code
```

The annotation `@spl.delta(ac)` registers the following function as a delta of `spl`, with the activation condition `ac` that follows the cross-tree constraint syntax. The function itself can have any name, but must have two parameters: the first one `variant` is the variant that is transformed by the delta; and the second one `product` is the product that may contain information necessary for the application of the deltas (e.g., the value of specific attributes). The transformation `code` performed by the delta is arbitrary python code. In particular, like in [54] transformation functions or methods must be provided to be able to construct a variant.

6.2 Application to CFD

Using the expressiveness available in feature models and in our python implementation in particular, we designed an initial feature model corresponding to a small part of the expected variability of a CFD tool. An except of that part is given in Figure 5. In particular, the `physical_model` option which represent

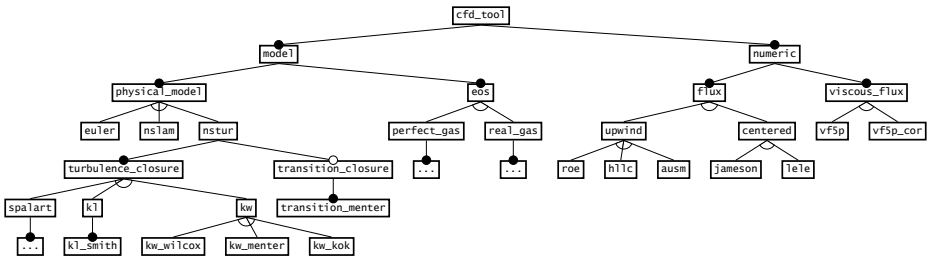

Fig. 5: Excerpt of our Feature Model

one aspect of the approximation method's variability is already quite large, and we didn't develop the `eos` subtree which too has many variation on the model of gas will be used in the fixpoint computation.

We illustrate our implementation of this feature model in Figure 6, with the implementation of the feature `turbulence_closure`. Note that like in Figure 5,

```
1 fm_turbulence_closure = FD("turbulence_closure",
2  FDXor(
3   FD("spalart", FDAnd(...)),
4   FD("kl", FDAnd(FD("kl_smith"))),
5   FD("kw",
6    FDXor(FD("kw_wilcox"), FD("kw_menter"), FD("kw_kok"))
7 )))
```

Fig. 6: Implementation of the Feature `turbulence_closure`

the three dots corresponds to a set of large subtrees.

Concerning the implementation of our variable operator specifications (VOSs), we implemented three core transformations on such specifications: the `add_input` method adds an input to the specification; the `add_output` method adds an output to it; and the `set_implementation` method states which implementation (given by the name of the implementing file) of the operator must be used.

We illustrate these methods in Figure 7, which presents the VOS of the `FxcUpwindMeanFlow` operator. Line 1 declares the VO. Lines 3–7 states that the

```
FxcUpwindMeanFlow = spl(fm, Operator)

@FxcUpwindMeanFlow.delta("upwind")
def fxc_upwind_construct_op(op, product):
 op.add_input(cfd_sig.conservatives(subsystem_term), cfd_sig.
    cell, vzone)
 op.add_input(cfd_sig.primitives(subsystem_term), cfd_sig.
    cell, vzone)
 op.add_output(cfd_sig.FxcUpwindMeanFlow, cfd_sig.cell, vzone
    )


@FxcUpwindMeanFlow.delta(And("perfect_gas", "roe"))
def fxc_upwind_mean_flow_perfect_gas_roe_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/roe"

@FxcUpwindMeanFlow.delta(And("perfect_gas", "hllc"))
def fxc_upwind_mean_flow_perfect_gas_hllc_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/hllc"

@FxcUpwindMeanFlow.delta(And("perfect_gas", "ausm"))
def fxc_upwind_mean_flow_perfect_gas_ausm_op(op, product):
 op.implementation = "fxc/upwind/mean_flow/perfect_gas/ausm"
```

Fig. 7: Implementation of the `FxcUpwindMeanFlow` variable operator

corresponding operator has two inputs and one outputs when the option `"upwind "` is selected. The fact that the operator has no input or outputs when `"upwind"` is not selected encodes the fact that this operator is no used in these cases. The actual implementation of the operator to use is state in the other deltas of the `FxcUpwindMeanFlow` product line. For simplicity, we only give the deltas related to the `"perfect_gas"` option, which all depend on which subfeature of `"upwind"` is selected.

7 Initial Results

To test our approach, we integrated it into a core running prototype that could run code on a single processing unit, either CPU, GPU or VE. We applied this prototype to a simple common usecase: the 2D NACA 0012 [42, 43]. This usecase is a simple 2D physical configuration modelling an airplane wing in a flow of air. The physical configuration is given in Figure 8: on the left, we have a input air flow modelling the plane going forward, in the middle, we have a wall modelling the cross section of the wing, and on the right we have the output flow. The zone where the air can flow is a disc, so not to introduce artifacts in the air flow caused by artificial angles.

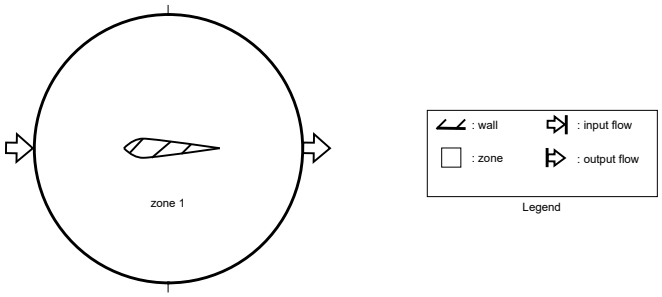

Fig. 8: Topology of the 2D NACA 0012 Usecase

The results of our study are shown in Figures 9 and 10. Figure 9 gives three convergence criteria of the fixpoint loop obtained by running three different configuration of our prototype: once configuring it to execute on a CPU, one on a GPU and the last one on a VE. It might not be obvious to see, but in this picture there are actually three red lines, three blue lines and three cyan lines, corresponding to the three criteria of the three runs of the prototype: three runs of our prototype, even if running on different hardware, are indistinguishable between each other.

Figure 10 shows the actual result of the equilibrium state computation done by our prototype. In particular, the picture on the left shows the equilibrium air density on a scale from blue (not dense) to red (dense); and the picture of the

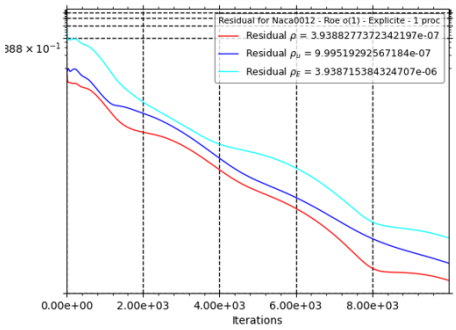

Fig. 9: Convergence of our prototype CPU/GPU/VE

right shows the equilibrium air speed, with arrows to show direction, and colour to show speed (blue being slow and red being quick).

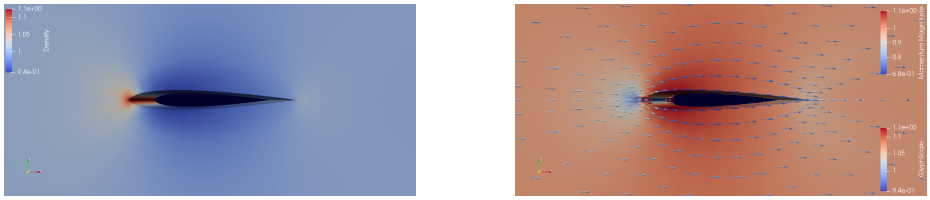

Fig. 10: Density on Vertex and Momentum on Vertex

8 Conclusion and Future Work

This work presents an study into the requirements of CFD tools, some limitations of the current tools available, elsA in particular. It then provided with an approach to solve some of these limitations. Similarly to several existing tools, this approach structures a CFD tool in three parts that distinguishes between: *i*) the operators that implement all the core mathematical function used in any computation; *ii*) the orchestrator that assemble the operators into a complete dataflow that computes the required data; and *iii*) the HPC layer which manages the distribution and concurrency during the computation of the dataflow. The novelty of our approach lies in the definition of the orchestrator part, which is based on tools originating from formal methods: *terms* and *order-sorted Algebra* are used to specify the inputs and outputs of the available operators, and pattern matching (a subcase of unification) is then used to identify dependencies between operators and generate the dataflow; *Delta-Oriented Programming* is

used to model the variability of the available operators, i.e., depending on the tool configuration, the inputs and outputs of the operators may change, which means that the dependencies between operators and thus the generated dataflow may change. Based on this approach, we implemented a prototype and tested it.

Going from a prototype to a useable tool still requires a lot of work. We need to integrate hardware distribution, and in particular integrate the possibility to manage heterogeneous hardware. A promising approach would be to use the standard API of `hwloc` [7, 20] that gives the topology of the hardware, with the characteristics of the different memory and processing nodes.

Another issue to tackle is memory allocation. Indeed, array of the right size must be allocated to host the data computed by the operators in the dataflow generated by our orchestrator. Because the dataflow can vary, so does the memory allocation. However, memory is allocated by hand in our prototype, and known approaches for memory allocation are not satisfactory: in pyFR and Devito, the memory is managed by the user directly; on the other hand, elsA does not have a general framework to model data and relies to enumerations to list all the possible data it can handle.

Another interesting aspect of this work is the similarities of the problem of generating the graph of operators and the problem of type inhabitance: as hinted in Section 4, the term modelling the data to compute is similar to a type, and from that point of view our generated graph is an expression that have that type. We will investigate this relationship further and possibly see if interesting result could be applied to our prototype. Moreover, this approach seems to integrate well with product lines. Indeed classic approach for product line definition is to add, remove or replace well identified code elements, but it is very difficult to have an expression always computing the same data in all variants, using however different methods to obtain it depending on the selected options.

References

1. Agosta, G., Fornaciari, W., Massari, G., Pupykina, A., Reghenzani, F., Zanella, M.: Managing heterogeneous resources in hpc systems. In: Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms. p. 7–12. PARMA-DITAM '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3183767.3183769
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.): Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY, Lecture Notes in Computer Science, vol. 12345. Springer (2020). https://doi.org/10.1007/978-3-030-64354-6
3. Apel, S., Kästner, C., Lengauer, C.: FEATUREHOUSE: language-independent, automated software composition. In: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. pp. 221–231. IEEE (2009). https://doi.org/10.1109/ICSE.2009.5070523
4. Aupoix, B., Spalart, P.: Extensions of the spalart–allmaras turbulence model to account for wall roughness. International Journal of Heat and Fluid Flow **24**(4),

454–462 (2003). https://doi.org/10.1016/S0142-727X(03)00043-2, selected Papers from the Fifth International Conference on Engineering Turbulence Modelling and Measurements
5. Biedron, R.T., Carlson, J.R., Derlaga, J.M., Gnoffo, P.A., Hammond, D.P., Jones, W.T., Kleb, B., Lee-Rausch, E.M., Nielsen, E.J., Park, M.A., et al.: FUN3D Manual: 13.7. National Aeronautics and Space Administration, Langley Research Center (2020)
6. Bourgeois, K., Robert, S., Limet, S., Essayan, V.: Geoskelsl: A python high-level dsl for parallel computing in geosciences. In: Shi, Y., Fu, H., Tian, Y., Krzhizhanovskaya, V.V., Lees, M.H., Dongarra, J., Sloot, P.M.A. (eds.) Computational Science – ICCS 2018. pp. 839–845. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-93713-7_83
7. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In: IEEE (ed.) PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. Pisa, Italy (Feb 2010). https://doi.org/10.1109/PDP.2010.67
8. Cambier, L., Heib, S., Plot, S.: The onera elsa cfd software: input from research and feedback from industry. Mechanics and Industry **14**(3), 159–174 (2013). https://doi.org/10.1051/meca/2013056
9. Ciżnicki, M., Kurowski, K., eglarz, J.W.: Energy and performance improvements in stencil computations on multi-node hpc systems with different network and communication topologies. Future Generation Computer Systems **115**, 45–58 (2021). https://doi.org/10.1016/j.future.2020.08.018
10. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.: Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In: Formal Methods for Eternal Networked Software Systems, Lecture Notes in Computer Science, vol. 6659, pp. 417–457. Springer International Publishing (2011)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, vol. 4350. Springer (2007)
12. Constantin, P., Foias, C.: Navier-stokes equations. University of Chicago Press (1988)
13. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Proceedings of International Software Product Line Conference (SPLC). vol. 3154, pp. 162–164 (07 2004). https://doi.org/10.1007/978-3-540-28630-1_17
14. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M.: A unified and formal programming model for deltas and traits. In: Huisman, M., Rubin, J. (eds.) Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10202, pp. 424–441. Springer (2017). https://doi.org/10.1007/978-3-662-54494-5_25
15. Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., Paolini, L.: Variability modules for java-like languages. In: Mousavi, M., Schobbens, P. (eds.) SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A. pp. 1–12. ACM (2021). https://doi.org/10.1145/3461001.3471143
16. Dick, A.J.J., Watson, P.: Order-sorted Term Rewriting. The Computer Journal **34**(1), 16–19 (01 1991). https://doi.org/10.1093/comjnl/34.1.16

17. Fay, M.: First-order Unification in an Equational Theory. Tech. Rep. 78-5-002, University of California at Santa Cruz (1978)
18. Flich, J., Agosta, G., Ampletzer, P., Alonso, D.A., Brandolese, C., Cappe, E., Cilardo, A., Dragić, L., Dray, A., Duspara, A., Fornaciari, W., Fusella, E., Gagliardi, M., Guillaume, G., Hofman, D., Hoornenborg, Y., Iranfar, A., Kovač, M., Libutti, S., Maitre, B., Martínez, J.M., Massari, G., Meinds, K., Mlinarić, H., Papastefanakis, E., Picornell, T., Piljić, I., Pupykina, A., Reghenzani, F., Staub, I., Tornero, R., Zanella, M., Zapater, M., Zoni, D.: Exploring manycore architectures for next-generation hpc systems through the mango approach. Microprocessors and Microsystems **61**, 154–170 (2018). https://doi.org/10.1016/j.micpro.2018.05.011
19. Focht, E.: Veo and pyveo: Vector engine offloading for the nec sx-aurora tsubasa. In: Resch, M.M., Kovalenko, Y., Bez, W., Focht, E., Kobayashi, H. (eds.) Sustained Simulation Performance 2018 and 2019. pp. 95–109. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-39181-2_9
20. Goglin, B.: Towards the Structural Modeling of the Topology of next-generation heterogeneous cluster Nodes with hwloc. Research report, Inria (Nov 2016), https://hal.inria.fr/hal-01400264
21. Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J., Winkler, T.: An introduction to obj 3. In: Kaplan, S., Jouannaud, J.P. (eds.) Conditional Term Rewriting Systems. pp. 258–263. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
22. Hähnle, R.: HATS: highly adaptable and trustworthy software using formal methods. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6416, pp. 3–8. Springer (2010). https://doi.org/10.1007/978-3-642-16561-0_2
23. Hähnle, R.: Task forces in the eternals coordination action. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6416, pp. 20–22. Springer (2010). https://doi.org/10.1007/978-3-642-16561-0_6
24. Hähnle, R.: The abstract behavioral specification language: A tutorial introduction. In: Giachino, E., Hähnle, R., de Boer, F.S., Bonsangue, M.M. (eds.) Formal Methods for Components and Objects - 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures. Lecture Notes in Computer Science, vol. 7866, pp. 1–37. Springer (2012). https://doi.org/10.1007/978-3-642-40615-7_1
25. Hähnle, R., Johnsen, E.B.: Designing resource-aware cloud applications. Computer **48**(6), 72–75 (2015). https://doi.org/10.1109/MC.2015.172
26. Hähnle, R., Schaefer, I.: A liskov principle for delta-oriented programming. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I. Lecture Notes in Computer Science, vol. 7609, pp. 32–46. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_4
27. Han, Z., Devarajegowda, K., Werner, M., Ecker, W.: Towards a python-based one language ecosystem for embedded systems automation. In: 2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP

and International Symposium of System-on-Chip (SoC). pp. 1–7 (2019). https://doi.org/10.1109/NORCHIP.2019.8906949
28. He, P., Mader, C.A., Martins, J.R.R.A., Maki, K.J.: Dafoam: An open-source adjoint framework for multidisciplinary design optimization with openfoam. AIAA Journal **58**(3), 1304–1319 (2020). https://doi.org/10.2514/1.J058853
29. Hink, R., Hannemann, V., Eggers, T.: Extension of the spalart-allmaras one-equation turbulence model for effusion cooling problems. In: Deutscher Luft- und Raumfahrtkongress 2013 (September 2013), https://elib.dlr.de/84638/
30. Hoefler, T., Schneider, T.: Optimization principles for collective neighborhood communications. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–10. IEEE (2012). https://doi.org/10.1109/SC.2012.86
31. Jasak, H.: Openfoam: Open source cfd in research and industry. International Journal of Naval Architecture and Ocean Engineering **1**(2), 89–94 (2009). https://doi.org/https://doi.org/10.2478/IJNAOE-2013-0011
32. Jung, Y.S., Baeder, J.: $\gamma - \overline{re_{\theta t}}$ spalart–allmaras with crossflow transition model using hamiltonian–strand approach. Journal of Aircraft **56**(3), 1040–1055 (2019). https://doi.org/10.2514/1.C035149
33. Kamburjan, E., Hähnle, R.: Deductive verification of railway operations. In: Fantechi, A., Lecomte, T., Romanovsky, A.B. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Second International Conference, RSSRail 2017, Pistoia, Italy, November 14-16, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10598, pp. 131–147. Springer (2017). https://doi.org/10.1007/978-3-319-68499-4_9
34. Kenway, G.K., Mader, C.A., He, P., Martins, J.R.: Effective adjoint approaches for computational fluid dynamics. Progress in Aerospace Sciences **110**, 100542 (2019). https://doi.org/10.1016/j.paerosci.2019.05.002
35. Khaleghzadeh, H., Manumachu, R.R., Lastovetsky, A.: A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms. IEEE Transactions on Parallel and Distributed Systems **29**(10), 2176–2190 (2018). https://doi.org/10.1109/TPDS.2018.2827055
36. Knopp, T., Eisfeld, B., Calvo, J.B.: A new extension for k–ω turbulence models to account for wall roughness. International Journal of Heat and Fluid Flow **30**(1), 54–65 (2009). https://doi.org/10.1016/j.ijheatfluidflow.2008.09.009
37. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: DeltaJ 1.5: delta-oriented programming for Java. In: International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14. pp. 63–74 (2014). https://doi.org/10.1145/2647508.2647512
38. Ladyzhenskaya, O.A.: Sixth problem of the millennium: Navier-stokes equations, existence and smoothness. Russian Mathematical Surveys **58**(2), 251–286 (apr 2003). https://doi.org/10.1070/rm2003v058n02abeh000610
39. Leicht, T., Jägersküpper, J., Vollmer, D., Schwöppe, A., Hartmann, R., Fiedler, J., Schlauch, T.: Dlr-project digital-x - next generation cfd solver 'flucs'. In: Deutscher Luft- und Raumfahrtkongress 2016 (February 2016), https://elib.dlr.de/111205/
40. Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P.A., Herrmann, F.J., Velesko, P., Gorman, G.J.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geoscientific Model Development **12**(3), 1165–1187 (2019). https://doi.org/10.5194/gmd-12-1165-2019

41. Mader, C.A., Kenway, G.K.W., Yildirim, A., Martins, J.R.R.A.: ADflow—an open-source computational fluid dynamics solver for aerodynamic and multidisciplinary optimization. Journal of Aerospace Information Systems (2020). https://doi.org/10.2514/1.I010796
42. McAlister, K.W., Carr, L.W., McCroskey, W.J.: Dynamic stall experiments on the naca 0012 airfoil. Tech. rep., NASA (1978)
43. McCroskey, W.: A critical assessment of wind tunnel results for the naca 0012 airfoil. Tech. rep., National Aeronautics And Space Administration Moffett Field Ca Ames ... (1987)
44. Meseguer, J., Goguen, J.A., Smolka, G.: Order-sorted unification. Journal of Symbolic Computation **8**(4), 383–413 (1989). https://doi.org/10.1016/S0747-7171(89)80036-7
45. Metcalf, M., Reid, J.K.: Fortran 90/95 explained. Oxford University Press, Inc. (1999)
46. Mofrad, M.H., Melhem, R., Ahmad, Y., Hammoud, M.: Graphite: A numa-aware hpc system for graph analytics based on a new mpi * x parallelism model. Proc. VLDB Endow. **13**(6), 783–797 (Feb 2020). https://doi.org/10.14778/3380750.3380751
47. Nielsen, F.: Introduction to MPI: The Message Passing Interface, pp. 21–62. Springer (02 2016). https://doi.org/10.1007/978-3-319-21903-5_2
48. NRC: Trace v5.0 theory manual - field equations, solution methods, and physical models. Tech. rep., United States Nuclear Regulartory Commission (2012)
49. Palacios, F., Colonno, M., Aranake, A., Campos, A., Copeland, S., Economon, T., Lonkar, A., Lukaczyk, T., Taylor, T., Alonso, J.: 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 (01 2013). https://doi.org/10.2514/6.2013-287
50. Perraud, J., Durant, A.: Stability-based mach zero to four longitudinal transition prediction criterion. Journal of Spacecraft and Rockets **53**(4), 730–742 (2016). https://doi.org/10.2514/1.A33475
51. Poinot, M., Rumsey, C.L.: Seven keys for practical understanding and use of CGNS. American Institute of Aeronautics and Astronautics (2018). https://doi.org/10.2514/6.2018-1503
52. Poirier, D., Allmaras, S., McCarthy, D., Smith, M., Enomoto, F.: The CGNS system. American Institute of Aeronautics and Astronautics (1998). https://doi.org/10.2514/6.1998-3007
53. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Bosch, J., Lee, J. (eds.) Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6287, pp. 77–91. Springer (2010). https://doi.org/10.1007/978-3-642-15579-6_6
54. Seidl, C., Schaefer, I., Aßmann, U.: Deltaecore - A model-based delta language generation framework. In: Fill, H., Karagiannis, D., Reimer, U. (eds.) Modellierung 2014, 19.-21. März 2014, Wien, Österreich. LNI, vol. P-225, pp. 81–96. GI (2014), https://dl.gi.de/20.500.12116/17067
55. Stroustrup, B.: A Tour of C++. Addison-Wesley Professional (2018)
56. Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T.: Featureide: An extensible framework for feature-oriented software development. Science of Computer Programming **79**, 70–85 (2014).

https://doi.org/10.1016/j.scico.2012.06.002, experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010)
57. Witherden, F., Farrington, A., Vincent, P.: Pyfr: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. Computer Physics Communications **185**(11), 3028–3040 (2014). https://doi.org/10.1016/j.cpc.2014.07.011
58. Young, V., Jaleel, A., Bolotin, E., Ebrahimi, E., Nellans, D., Villa, O.: Combining hw/sw mechanisms to improve numa performance of multi-gpu systems. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 339–351 (2018). https://doi.org/10.1109/MICRO.2018.00035
59. Zhang, N., Driscoll, M., Markley, C., Williams, S., Basu, P., Fox, A.: Snowflake: A lightweight portable stencil dsl. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 795–804 (2017). https://doi.org/10.1109/IPDPSW.2017.89