On the use of models for high-performance scientific computing applications: an experience report

Ileana Ober, Marc Palyart, Jean-Michel Bruel, David Lugato

On the use of models for high-performance scientific computing applications: an experience report

Ileana Ober[1] · **Marc Palyart**[2] · **Jean-Michel Bruel**[1] · **David Lugato**[3]

Abstract This paper reports on a four-year project that aims to raise the abstraction level through the use of model-driven engineering (MDE) techniques in the development of scientific applications relying on high-performance computing. The development and maintenance of high-performance scientific computing software is reputedly a complex task. This complexity results from the frequent evolutions of supercomputers and the tight coupling between software and hardware aspects. Moreover, current parallel programming approaches result in a mixing of concerns within the source code. Our approach relies on the use of MDE and consists in defining domain-specific modeling languages targeting various domain experts involved in the development of HPC applications, allowing each of them to handle their dedicated model in a both user-friendly and hardware-independent way. The different concerns are separated thanks to the use of several models as well as several modeling viewpoints on these models. Depending on the targeted execution platforms, these abstract models are translated into executable implementations by means of model transformations. To make all of these effective, we have developed a tool chain that is also presented in this paper. The approach is assessed through a multi-dimensional validation that focuses on its applicability, its expressiveness and its efficiency. To capitalize on the gained experience, we analyze some lessons learned during this project.

Keywords HPC · High-performance calculus · MDE · Model-driven engineering · Architecture · Fortran

Communicated by Prof. Dorina Petriu.

✉ Ileana Ober
ileana.ober@irit.fr

Marc Palyart
mpalyart@cs.ubc.ca

Jean-Michel Bruel
bruel@irit.fr

David Lugato
david.lugato@cea.fr

[1] IRIT, University of Toulouse, Toulouse, France

[2] University of British Columbia, Vancouver, BC, Canada

[3] CEA CESTA, Le Barp, France

1 Introduction

It is not every day that *Nature*—one of the most influential interdisciplinary research journals—focuses on topics related to computer science. In 2010, a paper dedicated to scientific computing [40] stressed the hard time scientists have due to the increased importance of software use and development in their work coupled with lack of formal training to software development. The paper revealed that scientific computer engineers face major problems in effectively maintaining scientific software, in a domain where software life spans count in decades, whereas the underlying hardware is highly volatile.

Since the 1960s, the increase in hardware performance has been continuous and followed Moore's law [44]. In this landscape, the last decade brought a major shift in the way hardware performance increases. While we were used benefiting from faster processor chips thanks to the decrease in transistors size, as we approach the physical limits of miniaturization, the increase in performance witnessed over the past 10 years is due to the widespread use of complex parallel architectures.

This change has some serious repercussions. Software applications no longer benefit automatically from performance increases. They need to be completely redesigned.

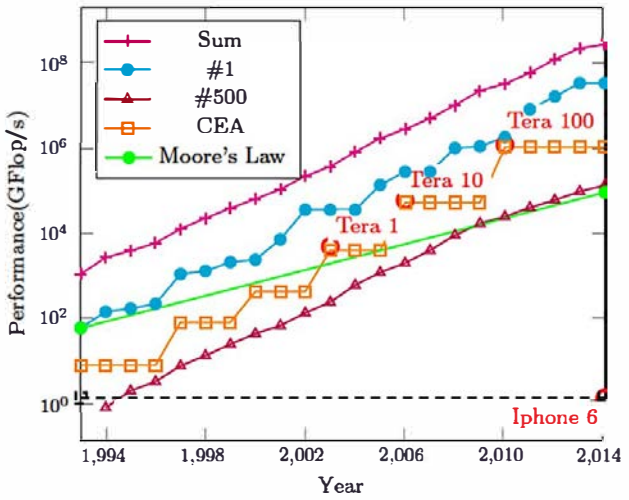


Fig. 1 Performance development

As highlighted in [38], the new performance gains mechanism *"shift[s] the burden of software performance from chip designers and processor architects to software developers."*

High-performance computing (HPC) has always been an area where parallel architectures were used to gain more computational power. Over the last 20 years, the performance of the world's most powerful supercomputers used by the HPC community has been scrutinized by the *TOP500* group.[1] Figure 1, generated via information available on Top500, illustrates that the performance increases in the last years. This chart highlights several interesting facts. First, by following the topmost curve (cross shaped and labeled *Sum*), we can see that the cumulative performance of the world's 500 most powerful supercomputers outpaces Moore's law. Moreover, we can see that the evolution is even more vigorous for the world #1 (dot shaped and labeled *#1*). This curve shows a more stepwise evolution, where steps correspond to technological breakthroughs that happen regularly and to the fact the top 500 is released twice a year. We must point out that these rapid hardware developments are also driven by energy performance goals. As highlighted in a recent study [55], performance growth shows a recent slowdown in performance increase, which is analyzed as being due to "hardware-, software- and funding-" related causes.

Since the world's most powerful supercomputer changes regularly, one may think that the strong performance increase would not correspond to the reality within the same organization. While it is true that the owner of the world's most powerful computer changes frequently, the need for performance increase within an organization is following a similarly strong increase, with all the software porting and maintenance issues that come with it. In Fig. 1, the square-shaped curve depicts the performance evolution within CEA, the French Atomic Energy and Alternative Energies Commission. As part of its Tera program [24], CEA plans to change its main supercomputer every 4 years. As one can see, this evolution follows the general tendency and outpaces Moore's law. This rapid growth in performance comes at the expense of inevitable hardware and software technological breakthroughs.[2]

1.1 Current practice in HPC application development

One of the main concerns of the high-performance scientific computing developer community is to produce efficient code for numerical simulation. Due to their thirst for computational power, the shift in the performance increase mechanism that involves the intensive use of parallelization had to be initiated a long time ago. As often highlighted (e.g., [54]), current mainstream parallel programming models, and in particular those addressing HPC, are low level and machine specific. Even though good performance levels can be achieved with these approaches, drawbacks in terms of architecture dependency, mix-up of concerns and programming complexity occur:

- Applications versus supercomputers lifetime cycle. Physics does not change frequently, supercomputers do. In this application domain, the life cycle of supercomputers is sometimes five to seven times shorter than the life cycle of scientific applications. For example, CEA's experience has shown that the simulation models and numerical analysis methods associated with our professional problems have a life expectancy of 20–30 years and must therefore be maintained over that period, with all the additional problems that come with software maintenance over such a period of time (e.g., team turnover).
- The lack of separation of concerns. The problem to be solved—the scientific knowledge of the physics—is entirely mixed with numerical schemes and target-dependent information, added to manage the parallelism. Once a complex system has been built, it is difficult to extract the physical models. As a result, maintenance and upgrading become even more complicated.
- Inaccessibility to domain experts. The complexity of software programming restricts the use of these workstations and supercomputers to a few scientists who are willing to spend a significant amount of time learning the characteristics of a particular set of machines to take full advantage of their capabilities.

Furthermore, the situation is getting worse with the new emerging generation of machines: hybrid machines. They are

[1] http://www.top500.org.

[2] It is instructive to note that the last iPhone6 could have been ranked in the top 500 in 1994, the reader may find the performance of his own computer in [21].

built by mixing heterogeneous hardware resources such as CPUs with many cores and graphics processing units. GPUs are usually found within graphics cards, where they compute the rendering of massive 2D and 3D scenes. However, hardware manufacturers of supercomputers have started to integrate GPUs, since they are particularly well suited to specific operations such as fast Fourier transform or matrix computations and thus linear algebra solving. GPUs contain a large number (in the range of hundreds) of stream processors which increase the computation power of supercomputers. To exploit them, however, developers have to depend on hardware manufacturer-specific instructions (NVIDIA Cuda [36]), or in the best case, on libraries which attempt to be more generic such as the OpenCL API [34].

Our research goal in this context is to apply model-driven techniques to define a tool-supported approach where the separation of concerns, the domain expert accessibility and reactivity to the hardware evolution would be addressed as primary concerns, not just as extra-functional features as they often are.

1.2 Overview of our approach

Our thesis is that model-driven development techniques can help us deal with the complexity existing in high-performance and scientific computing applications.

We feel that high-performance computing is a typical candidate for applying model-driven development techniques in order to deal with this complexity and to facilitate portability by using abstraction. This befitting is due to the fact there is a huge need for maintenance in terms of hardware architecture change; thus, techniques that help distinguish between platform-dependent and platform-independent code are much needed. This is precisely a strength of model-driven techniques.

Our approach consists in defining a method and a toolset supporting the use of the abstraction principles specific to model-based software development (MBSD) techniques in the context of HPC applications that use partial-differential equations with mesh-based numerical modeling. In order to assess the feasibility of this approach, we applied it to significant size HPC applications of realistic algorithm complexity, involving several domain expertises.

In this paper, we detail the following contributions:

- MDE4HPC—*our approach for adding abstraction in HPC applications* by applying the principles of MBSD. An earlier presentation of this approach was published previously here [51];
- An abstraction-based hierarchy of a *set of domain-specific languages* addressing the needs of various business domains, with a focus on *HPCML*—the domain-specific language that we defined as an answer to the abstraction needs of HPC applications. A partial overview of *HPCML* is available in a previous paper [52]. The current paper introduces the need for a hierarchy of domain-specific languages and positions *HPCML* in this hierarchy.
- A *toolset-ArchiMDE*—allowing the use of our techniques in an industrial setting. Our assumption is that we need an integrated tool set offering the functionalities needed by the various business domains (such as physicists, applied mathematicians), supporting *MDE4HPC* and natively using our domain-specific language *HPCML*, while offering services such as model transformation, code generation, validation and compiling. Although we partially describe the architecture of this tool set in previous publications [46,50,51], this paper gives a complete overview of our toolset.
- *A multi-dimensional validation* We have validated the approach by applying it to large HPC applications. Given the variety and complexity of HPC applications, we opted for a three-dimensional validation.

  First, we focus on the *overall applicability* of our method. For this, we used ArchiMDE to model a simplified Lagrangian hydrodynamic module. This case study, also described in [50], shows that it is possible to obtain viable code, in terms of performance, using *HPCML* and that this decreases the maintenance costs.

  The second case study aims at an *expressiveness assessment* on the capacity of our domain-specific language to model an application under industrial usage, based on a different physical domain. For this, we reverse-engineered a subset of a simulation software used in production for electromagnetism and show that it is possible to model it using our domain-specific language. This case study, essential in our view for a multi-dimensional evaluation, was never published before.

  The third validation aims at an *efficiency assessment*, and it is based on the second case study and consists in analyzing the execution time of the numerical simulation of the electromagnetism application code obtained by using our approach and comparing it with the original "old-style" code. For this, we have performed several sets of simulations on the Terra100 supercomputer, by varying the number of processors. These results have not been published before.

  This multi-dimensional validation shows that introducing abstraction in HPC applications is not incompatible with preserving performance. We consider this as a very important result in a domain culturally dedicated to low-level optimizations and closeness to hardware architecture.
- A set of *lessons learned* during our four-year experience in applying modeling techniques to high-performance computing. These lessons, which go beyond the context

of HPC, could be useful for further experiences of applying MDE to new domains.

The paper is organized as follows. Section 2 presents the main principles of our approach. The domain-specific language that we defined to address the abstraction needs of HPC is introduced in Sect. 3. Section 4 presents the tool chain that supports our approach. The multi-dimensional evaluation that we propose for our approach is detailed in Sect. 5 which also contains a broader discussion. Section 6 describes the lessons learned from this project. Section 7 presents related work on existing solutions for developing numerical simulations with a focus on approaches aiming at adding abstraction. Section 8 discusses the conclusions of our project and areas for future work.

2 MDE4HPC the step toward adding abstraction

As highlighted in the previous section, the major goal of our approach consists in increasing the abstraction level throughout the development life cycle of numerical simulation applications.

The key characteristics of high-performance computing applications are deeply analyzed here [22,32]. According to [22], the common characteristics of scientific application that makes use of high-performance computing are: (1) the computational performance involved, (2) the target range of platforms—and the changing targets, (3) the number of separate disciplines involved in the development teams, (4) the size of software effort to develop and build the needed software, (5) the size of software effort to maintain software and (6) the economical model. Our study explicitly addresses the characteristics (2), (3), (4) and (5) while concerned of the first one (1).

At first sight, the choice of a (subset of) UML could seem like a natural choice of language for our approach: It supports abstract reasoning with a good expressiveness and it is a fairly widespread language. However, in the context of scientific simulation software, the models expose information of various natures (numerical, physical, algorithmic, platform dependent, etc.) that is exploited by various types of domain experts. For this reason, instead of a large general modeling language, such as UML, we prefer to use a set of focused domain-specific modeling languages (DSML), ideally accessible to domain experts, as long as there is a set of model transformations to move from one DSML to the other. It soon became obvious that the definition of the DSMLs and transformations would need to go hand in hand with a dedicated tool suite and a set of methodological guidelines.

In this section, we introduce *MDE4HPC*, our approach for applying the principles of MBSD on the development of scientific simulation software. In particular, we describe the general structure of the domain-specific modeling language called *High-Performance Computer Modeling Language* (*HPCML*) on which the approach is built.

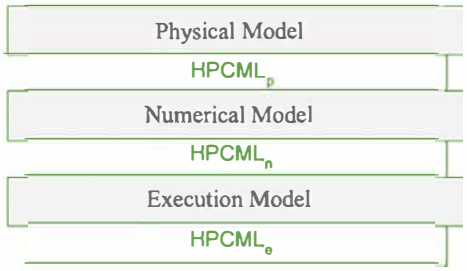


Fig. 2 Hierarchical abstraction layering in MDE4HPC

2.1 Overview of the approach

The *MDE4HPC* approach follows a classical model-based strategy: model all artifacts—including those that are non-software- dependent—produced by the range of experts who typically take part in the development (physicists, applied mathematicians, computer scientists, hardware and software architects) and combine them to generate the simulation code. To that end, the *MDE4HPC* approach offers a multi-layered architecture relying on several abstraction levels where each abstraction level targets specific types of experts.

Figure 2 depicts an overall view of the hierarchical abstraction layering in *MDE4HPC*. We find two types of elements in this diagram: models (one type per domain) and *HPCML* which acts as middleman between the three domains. For these reasons, *HPCML* is decomposed into three sub-languages: $HPCML_p$—the layer dedicated to physics, $HPCML_n$—the layer dedicated to mathematics and $HPCML_e$—the layer dedicated to execution-related matters (software and hardware). Several types of experts might be found in a given domain. Figure 3 illustrates the situation by showing the different viewpoints on the models of some domain experts and how they overlap. Each ellipse represents an abstract view of the model elements used by the corresponding expert as well as which part of the *HPCML* language they depend on. With this figure, we better understand that numerous artifacts are used and required by several domain experts. The *HPCML* language allows the existence of a common underlying model, thus facilitating the exchange of information between various domains. It also enables the traceability of decisions and concepts across the various abstraction levels.

2.2 Physics-dedicated layer

The physicist is usually the person formulating the requirements for a numerical simulation. He is also frequently the

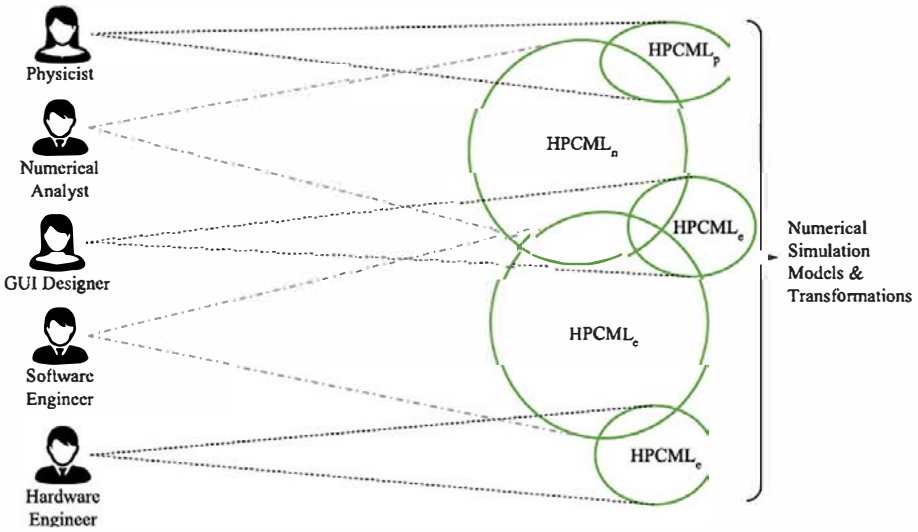


Fig. 3 Viewpoints of the domain experts taking part in the development

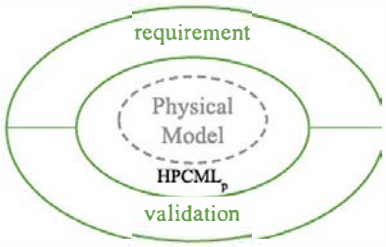


Fig. 4 $HPCML_p$—language dedicated to specification of the physical model

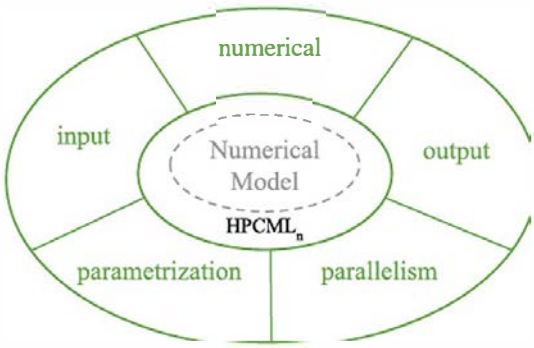


Fig. 5 $HPCML_n$—language dedicated to modeling the numerical information

final user of the simulation application. In our setting, the language $HPCML_p$ (p stands for Physics) provides primitives to support these two roles. Figure 4 depicts the structure of the $HPCML_p$ language with its two parts:

- *requirement* the physicist must be able to specify its physical model (in terms of equations, hypothesis, properties for example), as well as the modeling choices that led to this model.
- *validation* the physicist must be able to specify the validity range of its model and annotate this range with expected properties of the obtained results.

2.3 Mathematics-dedicated layer

$HPCML_n$ (n stands for Numerical) provides primitives required by a numerical analyst to precisely specify the numerical schema picked as a solution for the resolution of the physical model defined by a physicist. This part of $HPCML_i$s central to our approach, as it interacts with the two other parts of the language.

The primitives of $HPCML_n$ aim to abstract as much as possible all the software and hardware concerns related to the target execution platform for two reasons. First it improves accessibility for the numerical analysts who in general have a limited training in computer science. Second it increases the portability capability of the applications developed using our approach by making them less impacted by software and hardware technological breakthroughs. Overall the definition of this part of the language is not straightforward since we aim for a high abstraction level while being concrete enough to be able to specialize the numerical model to a concrete execution platform. Figure 5 provides an overview of the structure of the $HPCML_n$ language, whose major components are:

- *input*—the numerical analyst must be able to specify in an orderly manner input data for its simulation code. These parameters can either be of physical or mathematical nature. Typically, the information contained in this part of the model can be used to generate the graphical user interfaces that are used for exploiting simulation code.
- *numerical*—the numerical analyst must be able to specify her numerical algorithm using a formalism close to mathematics, which is her native domain-specific language.

This part of the language enables inter-application reuse thanks to composition mechanisms.

- *parallelism*—the numerical analyst expert must be able to specify potential sources of parallelism in her application without having to master advanced concepts specific to a supercomputer.
- *output*—the numerical analyst must be able to specify the computation results that need to be exported from the application. This part of the language provides voluntarily abstract concepts to facilitate integration with software used post-simulation such as scientific visualization software.
- *parametrization*—the numeric analyst must be able to specify how her application can be used for conducting parametric studies. This part of the model can be used to both generate user interfaces for conducting parametric studies and exploit the parallelism provided by such parametric studies.

2.4 Execution-dedicated layer

Our approach relies on a clear separation of concerns in order to allow each type of domain expert to focus on activities and information specific to their field. Thanks to the abstractions provided by $HPCML_n$, several computer-oriented aspects that in current practice are assigned to numerical analysts can be transferred back to hardware and software engineers.

The goal of the $HPCML_e$ (*e* stands for *Execution*) is to offer to hardware and software engineers the primitives needed to specify how a numerical resolution model specified with $HPCML_n$ should be implemented on a specific target platform. Figure 6 depicts the structure of $HPCML_e$, mainly focusing on three aspects:

- *optimization*—the engineer must be able to annotate a numerical model to specify what kind of optimization (improved algorithm, memory management, processor specificity, etc.) he wants to apply during the transformation toward a particular execution platform;
- *platform*—the engineer must be able to describe the execution platform. This specification would parametrize the model transformations leading to the simulation code generation;
- *transformation*—besides generic model transformations shared by several projects, the engineer must be able to define transformations specific to a particular project, either to enable some optimizations or to handle new platform characteristics.

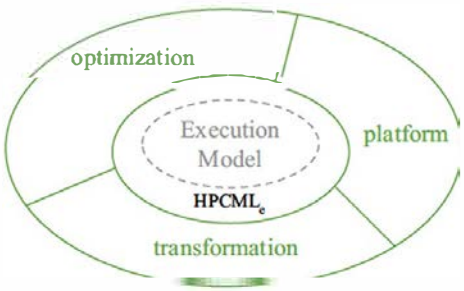


Fig. 6 $HPCML_e$—DSML that targets the software part

2.5 Development process

The development process promoted by MDE4HPC is presented in Fig. 7. We distinguish several models (oval shapes) based on the three sub-languages of *HPCML* we introduced in the previous sections. We notice that the *MDE4HPC* process can be mapped to a certain extent to the OMG Model-Driven Architecture recommendations [41]. An $HPCML_p$ model can be viewed as a Computation Independent Model

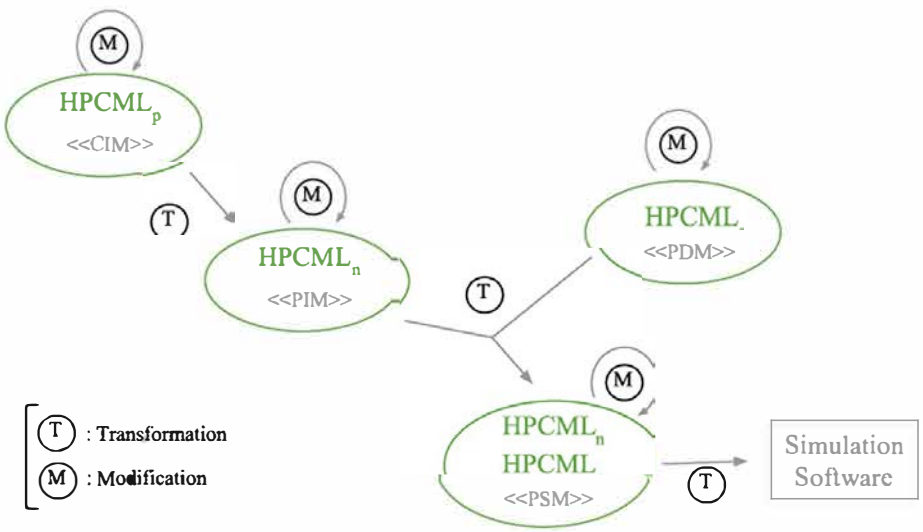


Fig. 7 MDE4HPC development process

(CIM) since it describes the problem. An $HPCML_n$ model can be viewed as a Platform-Independent Model (PIM) since it describes how to solve the problem. An $HPCML_e$ model describes the specificities of a target execution platform and thus can be viewed as a Platform Description Model (PDM). Finally, an $HPCML_n$ model specialized with an $HPCML_e$ model could be considered as a platform-specific model (PSM).

Overall, the *MDE4HPC* process is well decoupled as model elements updates can be performed regardless of their abstraction level. Thanks to the use of static model verification to check the conformance with the metamodel, it is possible to work on its own abstraction level without running the whole generation process. In addition, this process fits well with iterative and incremental development approaches that are adapted to the development of scientific software.

One of the major concerns that triggered our work relates to maintenance, in particular platform changes. In this situation, which results in adaptive maintenance, only the low-level steps of the process are affected: models and transformations based on $HPCML_e$. Concretely the platform-specific model and thus the application itself need to be regenerated as described in Fig. 7.

The layered definition of the *HPCML* enforced the separation of concerns, by allowing each domain expert to focus on artifacts relevant to its work. Obviously, the resulting layers are not disjoint (as one can infer from Fig. 3). The parts that are common to various domains are the ones that ensure the consistency of the transformations required when going back-and-forth between two domains. At the surface syntax level, the domain-specific language constraints proposed to various domain experts may even vary (in practice, this was seldom the case yet it occurred); however, as the underlying metamodel (for the concerned parts) is the same, the resulted transformation is accurate. As the various DSMLs target various aspects, a full transformation between these domain-specific languages would be irrelevant.

3 HPCML the dedicated language

Our work has been primarily focused on the definition of $HPCML_n$. For that reason, we usually refer to $HPCML_n$ solely as *HPCML*. This layer is located at the intersection of physical and execution domains ($HPCML_e$, $HPCML_p$), and it concentrates interesting scientific challenges. For example it covers concepts hard to express such as parallelism abstraction for which little experience has been capitalized by the modeling community. This section provides an overview of $HPCML_n$ abstract and concrete syntaxes. A more complete description of this language is available in previous publications [49,52].

3.1 Abstract syntax definition

The *HPCML* abstract syntax was defined through a metamodel based on the *ECore* metametamodel [57]. At its higher level, the *HPCML* metamodel is composed of six packages: *kernel*, *structure*, *behavior*, *output*, *validation* and *parametric*. The rest of this section focuses on the first three packages.

3.1.1 Kernel package

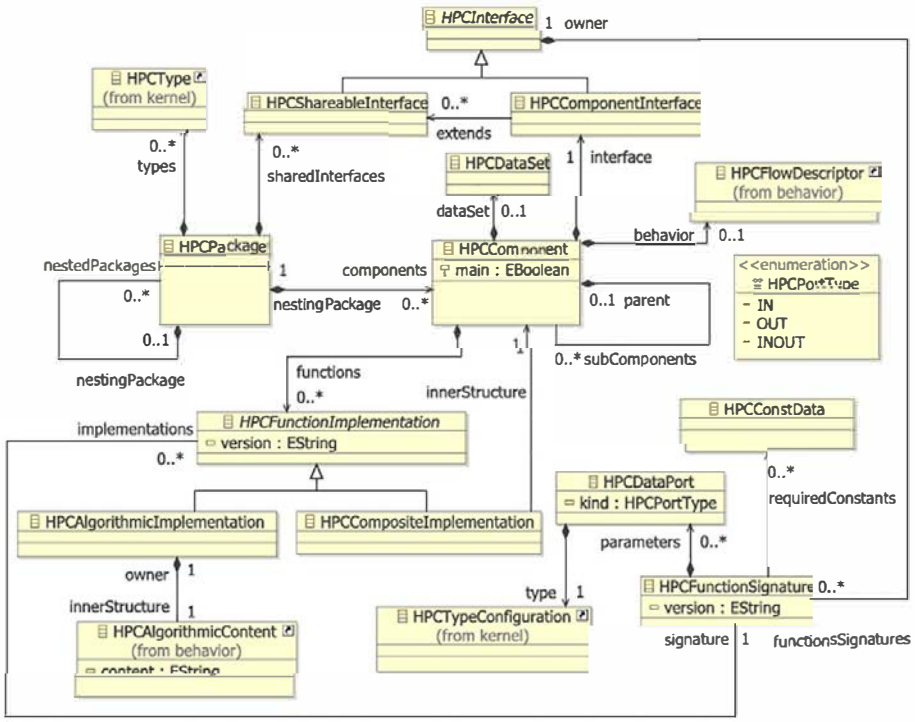
The *kernel* package gathers basic concepts that are used by the other packages. In particular it contains concepts regarding the definition of types and meshes. The *HPCML* language is provided with a library of standard model elements such as default types.

- *Basic types*—the higher-level concept. The standard library provides String and Boolean. Extension of basic types through inheritance is supported.
- *Numerical types*—the standard library provides usual numerical types (Integer, Real, Complex) but also provides domain-specific types (Length, Mass, Time, Volume, Speed, Frequency, Force, Pressure, etc.) in order to ensure strong typing in our type collection.
- *Structured types*—to assemble existing types, such as *Real3* composed of three reals to manipulate 3D coordinates.
- *Array types*—to model parameterizable arrays. The standard library provides *Matrix* and *Vector*.
- *Mesh related types*—probably the most important domain- specific data type, largely used in partial-differential equations and finite difference methods. The standard library provides *Nodes*, *Edges*, *Faces* and *Cells*, as well as structure using these kinds of elements.

3.1.2 Structure package

The *Structure* package of the meta-model (see Fig. 8) contains primitives to model the structural aspects of the application. It includes common structuring primitives, such as *Package*, *Component*, *Interface*, along with specific primitives, such as *DataSet*. Based on the observed need for structured sets of physical constants values, often used as tuning mechanisms of a numeric schema, we included in the meta-model the notion of *DataSet*, which models sets of strongly typed data, built upon custom base types. The *DataSet* concept was introduced to make sure that physical constants are managed properly, i.e., that their values are consistent over the whole application.

Fig. 8 Package *structure* from *HPCML*$_n$



3.1.3 Behavior package

One of the central packages in our meta-model is the *Behavior* package presented in Fig. 9. It gathers behavior-specific building blocks to define the flow of an application by composing and referencing structural building blocks. The current practice in the development of HPC applications oriented us toward a formalism inspired by UML and SysML like *activity diagrams*, because they offer a convenient mix of data flow and control flow. Accordingly, this package allows us to model:

- *Control flow*—to model the control graph of the application in terms of oriented graphs formed of nodes (behavior-intensive activities) and transitions (to model the behavior flow between nodes).
- *Data flows*—to model the data flow within an application. The specification is done in terms of functions called and their flow. The functions are connected using *communication ports*, which offer means for conveying variables issued from some data set, parameters of the current data flow or some local variables.
- *Aspects*—to facilitate the separation of concerns between the main numerical model and side concerns such as numerical validation or output management, we introduced concepts (e.g., `HPCAspect`) that support an aspect-based approach [35].

Given its target domain, our DSML needed parallel programming- specific primitives. The definition of these primitives is inspired by existing parallel design patterns [33] and in-house libraries using parallel programming. Our metamodel contains the following main primitives to model parallel aspects:

- *Task parallelism*—to model parallel tasks. `HPCFork` and `HPCJoin` primitives serve to achieve the modeling of this parallelism.
- *Domain decomposition* (`HPCSetEnumerator`)—to decompose the domain on which the computing is performed into sub-domains, so that each of these sub-domains may be assigned to a distinct physical execution unit. In a real setting, performing the computation on a grid element requires access to (connex) grid elements. In such a case, the dependencies are explicitly captured at the model level and can be exploited during code generation, for instance by defining code generation patterns that make local copies with dummy grid elements, in order to minimize the resource- consuming communications.
- *Parallel loops* (`HPCParallelFor`)—to model parallel iterations.

To capture the variety in terms of dependencies between iterations in a parallel loop, we introduce the possibility to distinguish between two kinds of iterations. On the one side, there are iterations without outputs between iteration steps (but with possible access to the same data), with accumulation of the iteration (which therefore requires a reduction operation in order to recombine the result after each iteration). On the other side, there are more complex iterations

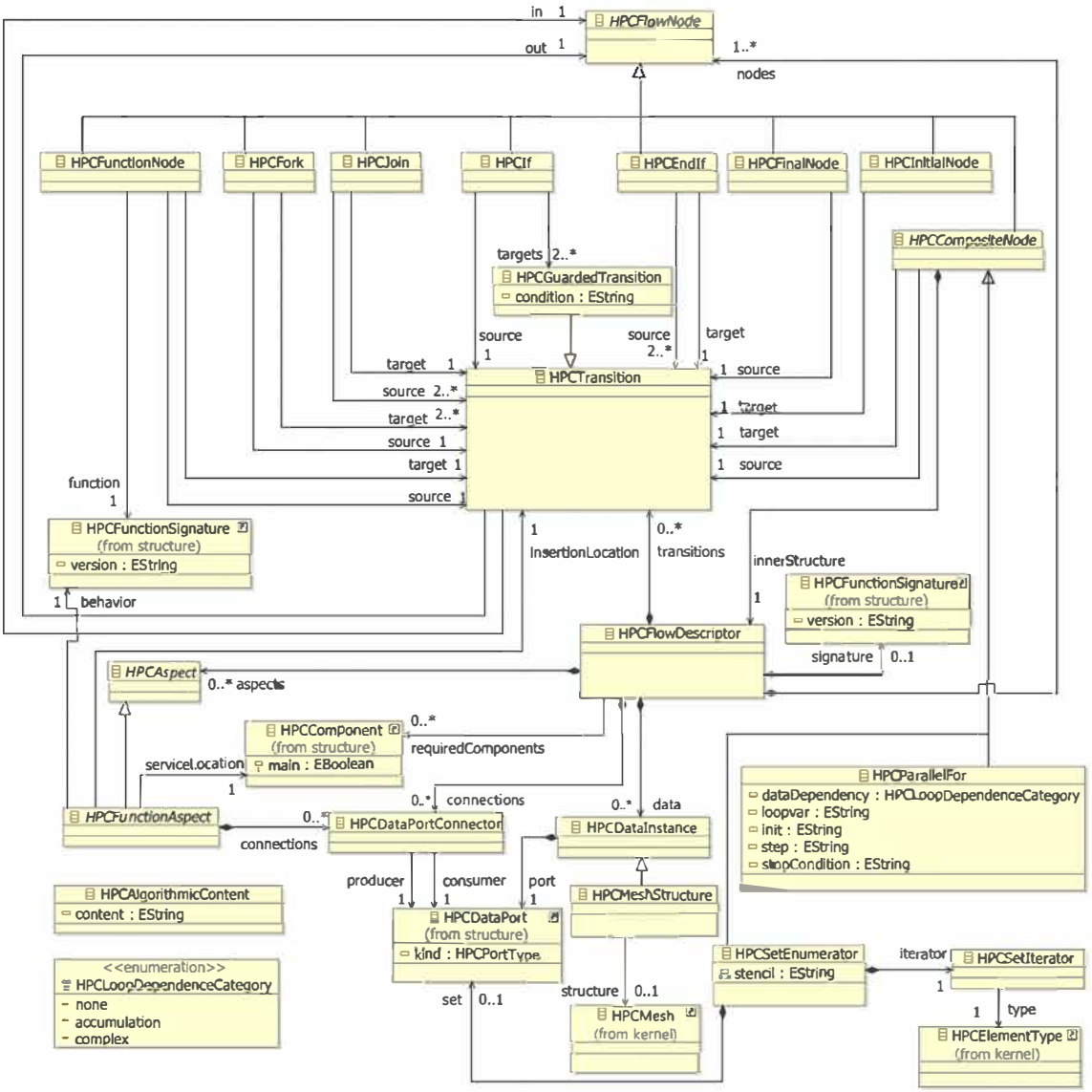


Fig. 9 Package *behavior* from $HPCML_n$

where the dependencies between iteration steps go beyond simple data accumulators. Capturing the information on the kind of iteration is critical when generating code and deciding on the application deployment.

Let us stress here that, as in the case of any language definition, the primitives presented here do not preclude their bad usage. In particular, in the case of task parallelism expressed at model level, we need to check afterward for possible deadlocks or possible race conditions. One possibility would be to use symbolic execution libraries.

3.1.4 Others packages

In addition to the three packages, we have presented *HPCML* that contains three other packages:

The *output* package provides concepts for specifying in an abstract way outputs of a simulation code. Output data of a simulation code can be gigantic. Different storage formats and ways of collecting outputs exist. Abstracting these aspects allows us to adapt easily to all these options.

The *validation* package provides concepts to specify properties of model elements such as post-conditions, preconditions or test methods based on aspect-oriented techniques.

The *parametric* package offers all the concepts required to design parametric studies. The idea here is also to benefit from the large amount of parallelism that can be extracted from these studies and to generate tailored simulation applications to exploit it.

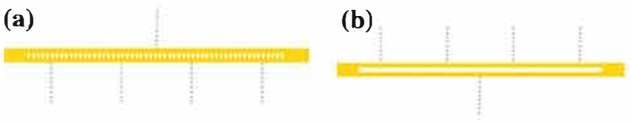


Fig. 10 Concrete syntaxes for Fork and Join in *HPCML*. a HPCFork. b HPCJoin

3.2 Concrete syntax definition

This section gives an insight into the concrete syntax definition of *HPCML*. The concrete syntax definition is often a rather neglected issue, probably because presenting it requires a lot of lengthy uninteresting details. Without going into a throughout syntactic description, we present here some principles that guided our choices.

The graphical syntax of a modeling language is a very important premise for its adoption. As stressed by [43], "research in diagrammatic reasoning shows that the form of representations has an equal, if not greater, influence on cognitive effectiveness as their content." Aware of the importance of the graphical representation, we designed it by applying Moody's theory [43] which encompasses nine general principles to conceive cognitively efficient graphical notations. These principles range from the need for perceptual discriminability and semantic transparency to the need for a cognitive integration (for a detailed overview of these principles, the reader is referred to the paper [43]).

To illustrate the impact of applying this theory on the graphical syntax of *HPCML*, let us refer to the *join* and *fork* concepts. Figure 10 illustrates comparatively the symbols representing these concepts. The symbols we have chosen are inspired by symbols frequently encountered in many languages to represent synchronization (e.g., UML activity diagrams). Contrary to these other languages that use the same symbol for both concepts, we chose to graphically differentiate them while also improving their semantic transparency (principles 2 and 3 in [43]). The dashed lines represent potential incoming and outgoing transitions. The inner part of the fork is composed of several small blocks expressing the decomposition into several execution flows. On the contrary, the inner part of the join is composed of one unique block expressing the merging of the different execution flow into one.

Another example of our design choices in terms of concrete syntax is illustrated in Fig. 11. The *HPCSetEnumerator* symbol uses a rectangular shape to recall the drawing zone as it can contain an inner structure similar to the overall diagram. The icon present at the top left part of the diagram suggests the decomposition of a domain (the big square) into sub-domains (the four small inner square).

The right side of Fig. 11 illustrates the symbol for an *HPCParallelFor*. Its graphical representation is intentionally close to the one of *HPCSetEnumerator*, as the two concepts are semantically close and are derived from the same concept—the *HPCCompositeNode*. The icon present at the top left zone is different at this time and suggests the parallel semantic of a *HPCParallelFor*.

This section presented *HPCML*, the domain-specific modeling language dedicated to high-performance numerical simulation applications. We looked at both the abstract and concrete syntax of this language. Section 5 will give another glimpse of this language through some examples. Before that, we will focus in the next section on an overview of the tooling we have developed to support the *MDE4HPC* approach and *HPCML*.

4 ArchiMDE the tool

Only a true integrated development environment (IDE) bringing together all the services needed by the various skills taking part in development would be in a position to respond to the requirements of the *MDE4HPC* approach. Such an IDE would, among other things, have to manage modeling activities for the various skills, for the transformation of models, validation, code generation, compilation, debugging and version management.

This section presents the main outlines of the *ArchiMDE* (pronounced ar-shi-med) tool, an IDE implementing $HPCML_n$ language and following the recommendations of the *MDE4HPC* approach. After a presentation of its software architecture and the Paprika tool that it includes, we detail

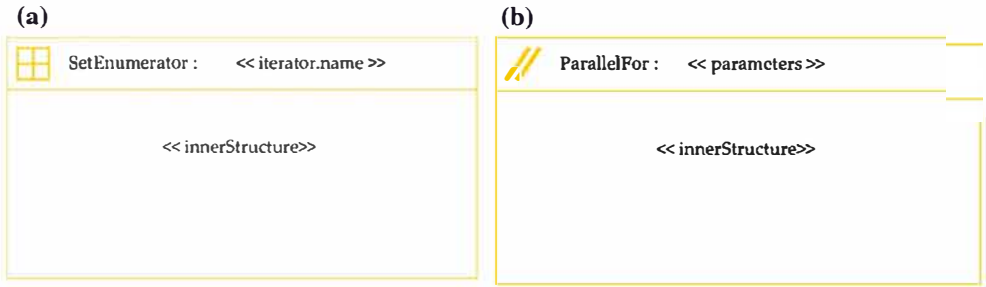


Fig. 11 Concrete syntax for set enumerator and parallel for in *HPCML*. a HPCSetEnumerator. b HPCParallelFor

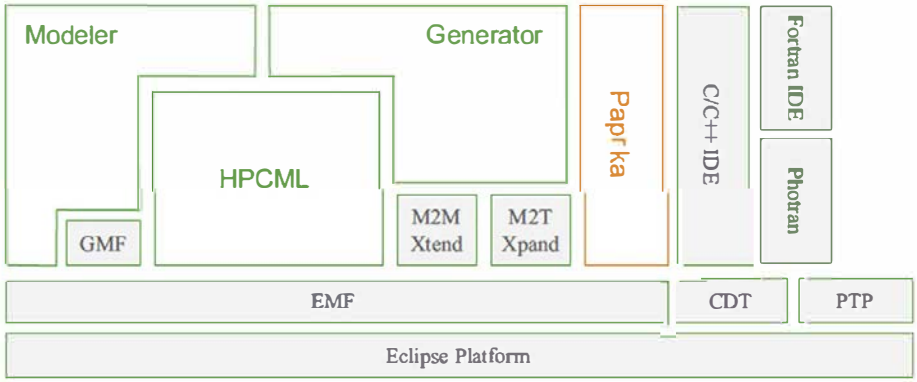


Fig. 12 Software architecture of *ArchiMDE*

the development process associated with it that leads to the generation of applications based on the *Arcane* [25] platform. Arcane is a software development framework for 2D and 3D numerical simulation codes that handles the mesh management and the parallelism strategy. We also address problems of implementing integration of low-level algorithmic with models.

One advantage of *MBSD* is that there exists a whole panel of tools enabling you to implement approaches based on it. Among this panel, you find tools that more or less faithfully implement OMG standards, like the set of projects under the *Eclipse Modeling Project* [57].

Our choice went in favor of the *Eclipse* platform [17] as it has a number of interesting characteristics in addition to its predispositions for MBSD through the *Eclipse Modeling Project* that focuses on the evolution and promotion of model-driven development technologies. At its core, the *Eclipse Modeling Framework* (EMF) [57] provides a metamodeling language known as *ECore*. *ECore* is a metamodel whose specification comes close to the *essential* MOF standardized by the OMG [27]. The implementation of the *HPCML* metamodel within *AchiMDE* is based on *ECore*.

Figure 12 offers a simplified view of *ArchiMDE*'s software architecture. In addition to the components we have just mentioned, there are components that take charge of model transformations (*Xtend*, *Xpand*) as well as the Graphical Modeling Framework (*GMF*) that provides a development approach based on models to implement the concrete syntax of a metamodel. In what follows, we describe the *Paprika* component in detail.

4.1 Paprika studio

During previous works, we have already used *MBSD* to improve the development of numerical simulation software thanks to the *Paprika* tooling [46]. This tool aims to facilitate the creation and maintenance of GUI dedicated to the editing of data sets used to parametrize simulation codes.

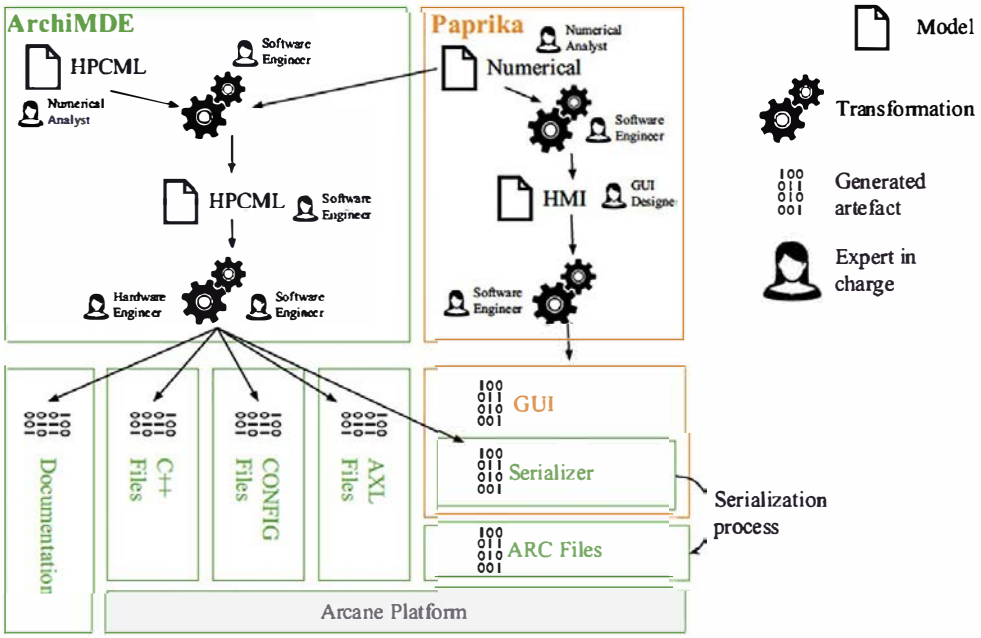
The use of *Paprika* relies on a two-stage process. The first stage of a *Paprika* development is the modeling of the structure and the constraints specific to the data set required by the target simulation application. This step is performed according to the *Paprika:Numerical* metamodel which was defined with the *ECore* metamodel. Its complete definition is available in [46]. The two basic concepts of this metamodel are types (*Type*) and data (*AbstractData*). Complex types can be constructed by combining or extending the simple basic types provided by *Paprika*. One of the special features of this metamodel is that it provides a strong typing system: An existing type can be combined with a physical unit to create highly specific data types.

Once the *Numerical* model has been defined, the second stage of the process offers two possibilities. In the first case, *Paprika* can generate a GUI based either on the Google Web Toolkit (*GWT*) [28] or on the Standard Widget Toolkit (*SWT*) [47] allowing the final user to create and edit data sets respecting the structure and constraints defined in this model from the previous step. In the second case, *Paprika* provides a generic editor capable of editing data sets from any type of *Numerical* model. Nassiet et al. [46] provide further details on these two approaches, especially regarding their advantages and drawbacks. Overall, *ArchiMDE* reuses and integrates *Paprika* thanks to a set of model transformations.

4.2 Tool operation

The internal process of the *ArchiMDE* tool is shown in Fig. 13 through the sequencing of the various transformations of models on which it is based. We provide several points of view, also known as perspectives (see also Fig. 3), on the models adapted to the various skill profiles of the participants in the development of a simulation application: Applied mathematicians are able to take advantage of perspectives to implement a numerical schema as independently as possible from the platform; GUI developers benefit from perspectives for the management of inputs/outputs of the simulation

Fig. 13 *ArchiMDE* transformation process



code; finally, through transformations, software developers determine how abstract models designed by applied mathematicians can be translated into executable code for a target platform.

Regarding software target platform, the current version of *ArchiMDE* is capable of generating applications based on the *Arcane* framework [25]. Figure 13 shows all the software artifacts needed to create an *Arcane* application and how they are generated via model transformations. Generally, there is duplication of data between the source code of a simulation code and the associated graphic interface to edit its data set. As the latter are most often written in different programming languages, a serialization mechanism must be set up to establish communication between the two applications. As the *Paprika* tool does not know how these data sets must be generated, it does not allow the data set editor serializer to be generated automatically. Although this stage is elementary, it was always done manually. Now *ArchiMDE* has a global vision of the application and can thus generate this serializer. Thanks to that serializer, ARC configuration files (data set format for *Arcane*) can be created (see Fig. 13). This example clearly shows that *ArchiMDE* allows the various developer profiles to focus on their core skill while also facilitating the exchange and reuse of data between the various skills.

Although Fig. 13 represents the sequencing of the various transformations more or less sequentially, a frequently used technique of iteration on the first transformations can also be adopted (*HPCML* merged with *Paprika:Numerical*) without using lower-level transformations (*HPCML* to *Arcane* or *Paprika:Numerical* to *Paprika:HMI*). This flexibility during development complies with the recommendations of the *MDE4HPC* approach.

4.3 Low-level algorithmic management

The *HPCML* version presented here does not offer a formalism to define an `HPCAlgorithmicContent` but just the constraints on the formalism to be used. There are several ways in *ArchiMDE* to manage integration of the low-level algorithmic (expressed in the formalism provided by the *Arcane* platform) with high-level models. Indeed, we have identified four generation management approaches: direct, incremental, algorithmic model and synchronization. For further details regarding the advantages and drawbacks of each approach, see [49].

As our objective was to start by defining and validating the $HPCML_n$ language, which has a considerable impact on the development of the tool, we have selected the direct generation approach. In this configuration, the target platform language (here *Arcane*) is used to describe the `HPCAlgorithmicContents`. A *String* attribute is used to store the content. On generation, this content is used as such by model transformations.

Although it is only a prototype, this tool was used as a medium for practical works at the international summer school organized by the CEA, EDF and *Inria* in 2010. This event, the topic of which was "*Sustainable High Performance Computing*", provided the opportunity to have the tool tested with more than 40 people. The considerable feedback on missing or unsuitable concepts contributed to upgrading the *ArchiMDE* tool as well as the $HPCML_n$ language.

5 Evaluation and discussions

In this section, we present the evaluation we have performed for our approach and we take advantage of the conclusions raised by this evaluation to more largely discuss on *MDE4HPC* in Sect. 5.4. The evaluation of the *MDE4HPC* approach was performed in two stages. First we used the *ArchiMDE* tool to redevelop the simplified Lagrangian hydrodynamics code introduced in [25]. The evaluation of this development, the results of which have been published in [50], enabled us to validate the fact that it is possible to use the *MDE4HPC* approach to generating viable code in terms of performance while obtaining a reduction in maintenance costs. Secondly we wanted to evaluate the capacity of $HPCML_n$ language to model an application both used in production and based on another physical domain. This section presents the results of this new evaluation.

5.1 Presentation of the simulated system

The simulation code introduced in this section is based on a code used in production that serves to simulate the diffraction of a plane electromagnetic wave by complex 3D objects composed of several materials. These materials may be conducting or dielectric. In our case, we focused on conducting mono-material 3D objects.

5.1.1 Problem

A perfectly conducting body is illuminated by an incident harmonic plane wave. The presence of this obstacle modifies the incident field: This is what is known as the diffraction phenomenon. Our goal is to compute the electric current induced by this incident wave on the object's surface.

5.1.2 Physical model

The physical model chosen for the study of this phenomenon is based on Maxwell's equations [29].

5.1.3 Mathematical models for resolution

For the resolution, we rely on a formulation based on the surface integral equations of Maxwell's equations in harmonic regime. We thus reduce a three-dimensional problem on an unbounded domain to a boundary problem posed on a surface.

The integral equations in a variational formulation that we propose to resolve are the conventional electric-field integral equations (EFIE) [18]. We then use the finite element method to solve these integral equations. The main stages in constructing a finite elements model are as follows [48]:

- discretization of the continuous medium into sub-domains (meshing);
- construction of the approximation by sub-domain (choice of the finite elements);
- computation of the elementary matrices corresponding to the variational formulation of the problem;
- assembly of elementary matrices;
- consideration for boundary conditions;
- resolution of the system of linear equations of type $\mathbf{Ax} = \mathbf{b}$.

5.2 Modeling of the application with HPCML

5.2.1 Modeling process

The existing code, from which the models presented in this section are derived, is written in *Fortran 90* and uses the Message Passing Interface *MPI* [56] standard (Message Passing Interface) [56] for expressing parallelism. The rise of the abstraction level when (retro)modeling was performed could only be achieved with participation from mathematicians who passed on their knowledge during more or less formalized discussions. Even if the approach adopted (retrodesign) in this case study is not identical to a new development, we were able to identify a general process that seemed to be suited to both cases.

The first stage of this process, which takes place during discussions on the mathematical model, involves drawing on a sheet of paper the hierarchy and sequencing of functions using a simplified version of the *HPCML* concrete syntax. Once this first rough sketch is completed, we can determine what functions are available in the existing components. Then we can model the application by following the sequence described in the rough sketch we just established (sequencing of functions and their origins). Finally, a global refactoring of a subset of the application is performed to determine whether all the potential sources of parallelism have been identified.

5.2.2 General overview of the models

Figure 14 presents a high-level structural view of the application by showing the organization of the main components. Two main packages (`HPCPackage`) can be seen: The first one contains the components specific to the simulation code we are developing (*EMPrototypeMDE Project*), and the second contains components providing generic numerical services (resolution of linear system and integration).

Figure 14 also shows part of the structure of the application's main component (`HPCComponent` *EMPrototypeMDE* with the `main` property *true*) and especially the definition of its interface (`HPCComponentInterface`). It is interesting to note that even on the first hierarchi-

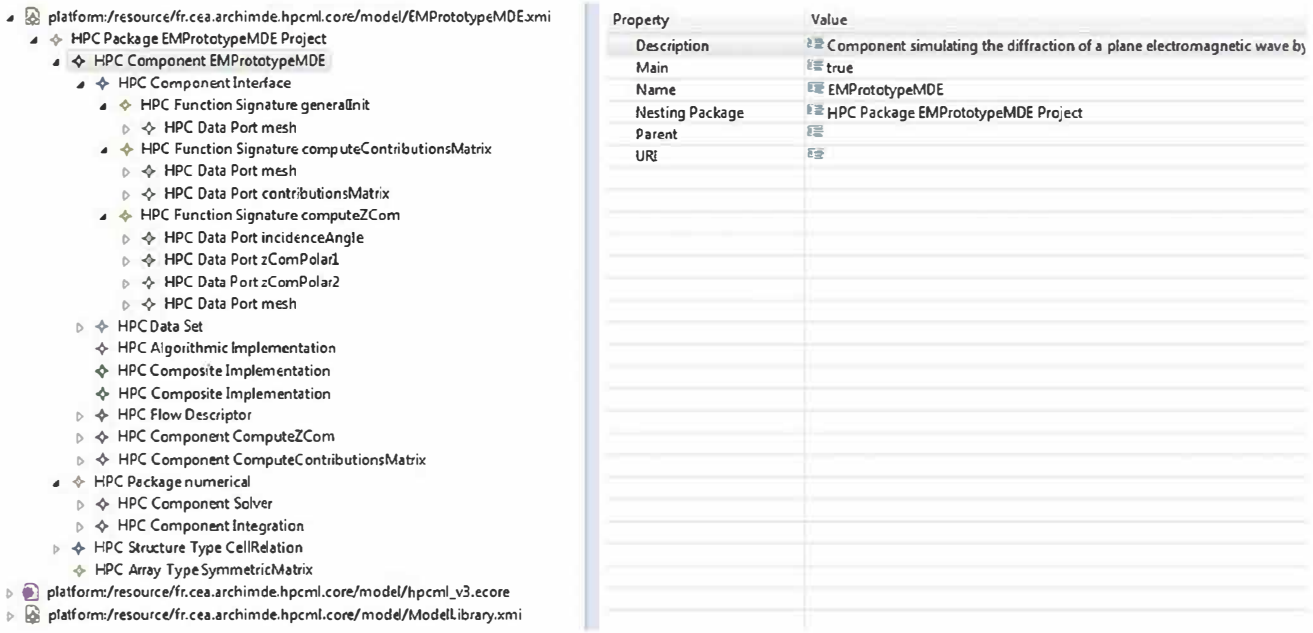


Fig. 14 Main view of the structural part

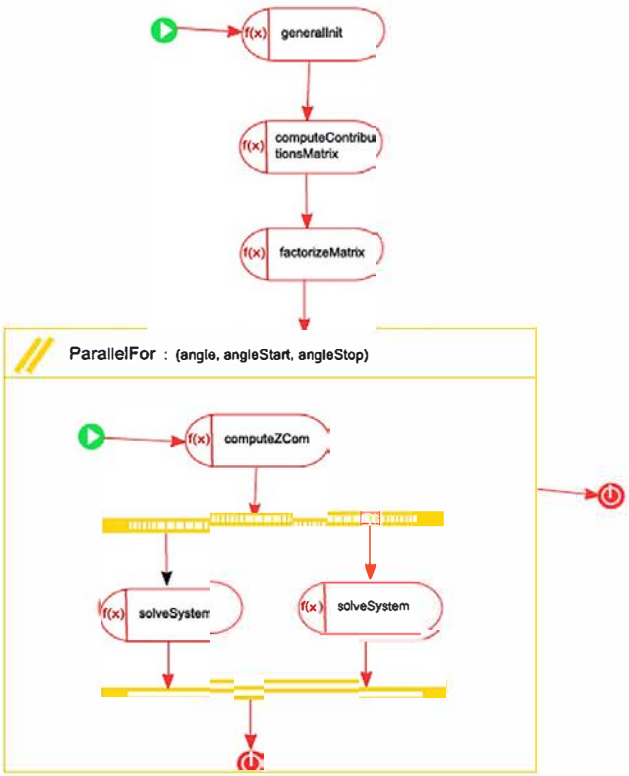


Fig. 15 `HPCFlowDescriptor` of the *EMPrototypeMDE* component

cal level of the *EMPrototypeMDE* application component, you see both sorts of implementations (`HPCComposite Implementation` and `HPCAlgorithmicContent`).

The structural view follows the *master/details* UI pattern with two zones: a tree on the left representing the hierarchy of the model elements and on the right the details of the selected model element.

Concerning behavior modeling of the application, Fig. 15 shows the `HPCFlowDescriptor` of the *EMPrototypeMDE* component where only the execution flow is visible. An `HPCFlowDescriptor` allows to specify the execution flow of an application or a function with a semantic close to activity diagrams in UML. This one uses several forms of expression of parallelism: an `HPCParallelFor` (same processing applied to different data) and the couple `HPCFork`/`HPCJoin` (different processing concurrently). Note that the interleaving of several constructions to express parallelism allows them to be combined to obtain even more parallelism.

5.2.3 Calculation of the matrix of contributions

Figure 16 gives a view of how the *ComputeContributionsMatrix* component, which is a sub-component of the *EMPrototypeMDE* component, is used to refine the behavior specification of its parent.

Meanwhile Fig. 17 shows the `HPCFlowDescriptor` of this component with its control flow. Some new elements as compared to the `HPCFlowDescriptor` of the *EMPrototypeMDE* component present in Fig. 15 can be distinguished here.

First the *ComputeContributionsMatrix* component is a sub-component of the *EMPrototypeMDE* component, and

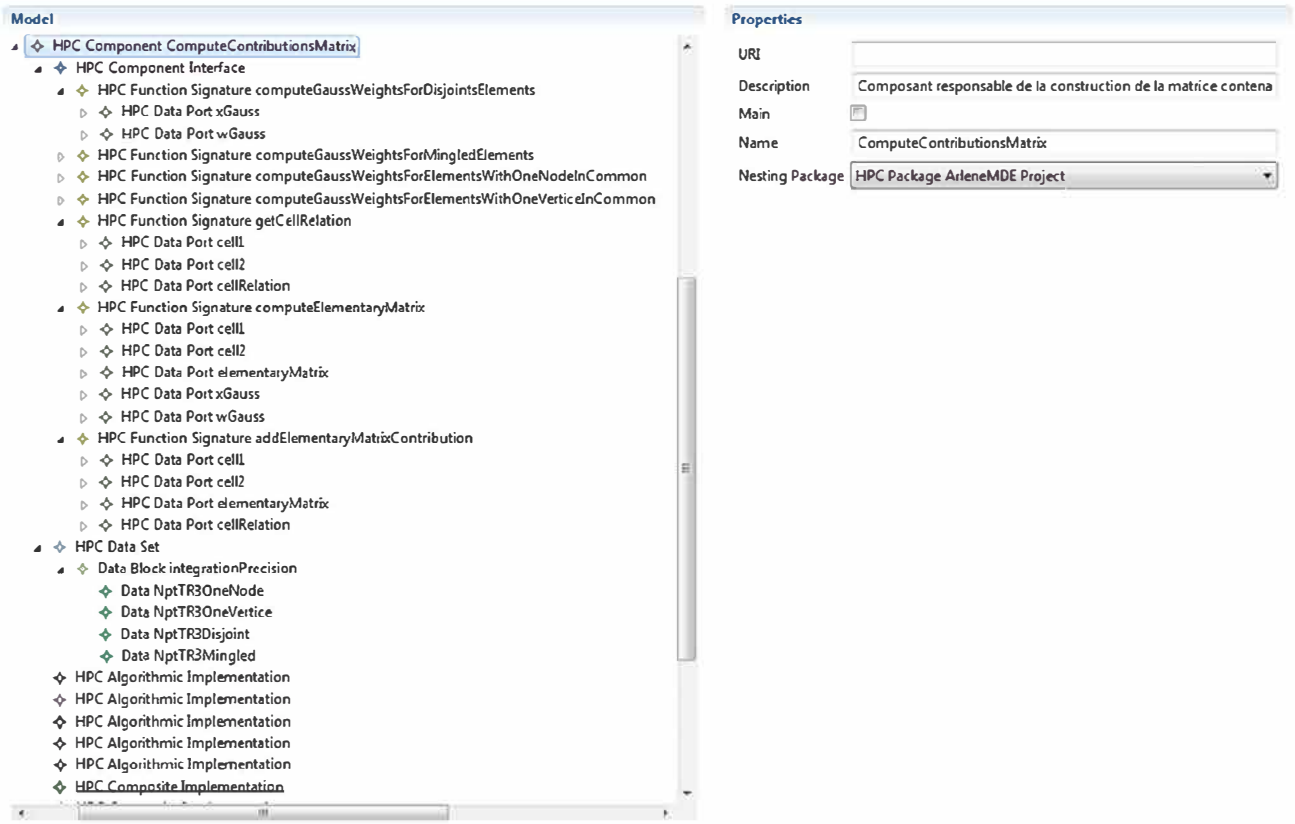


Fig. 16 View of the structural part of the *ComputeContributionsMatrix* component

as a result, its `HPCFlowDescriptor` has an `HPCFunctionSignature` that results from one of the `HPCFunctionSignatures` of the *EMPrototypeMDE* component interface: It is the *computeContributionsMatrix* signature whose declaration is visible in Fig. 14.

Second you can see an example of the third way of expressing parallelism: the `HPCSetEnumerator`. Here we even find the interleaving of two `HPCSetEnumerators` since we wish to calculate the contribution of each mesh with respect to the other meshes. It is not visible on the figure since it is not defined through this view, but the value of `stencil` (the model element describing the memory dependencies) for these two `HPCSetEnumerators` is *all*, since each mesh will have to access all the other meshes, at least in read (`HPCDataPort` of the relation set in mode `IN`). This is typically a case that will pose more problems on generation of implementation toward architectures with distributed memory than on those with shared memory. In the case of shared memory, it can be imagined that all the execution units access mesh data by their common memory. In the second case, a number of solutions can be envisaged one of which would be to replicate the meshing data on the memory of each execution unit. Of course, this solution can only be considered if the memory of the execution units is big enough to receive mesh data.

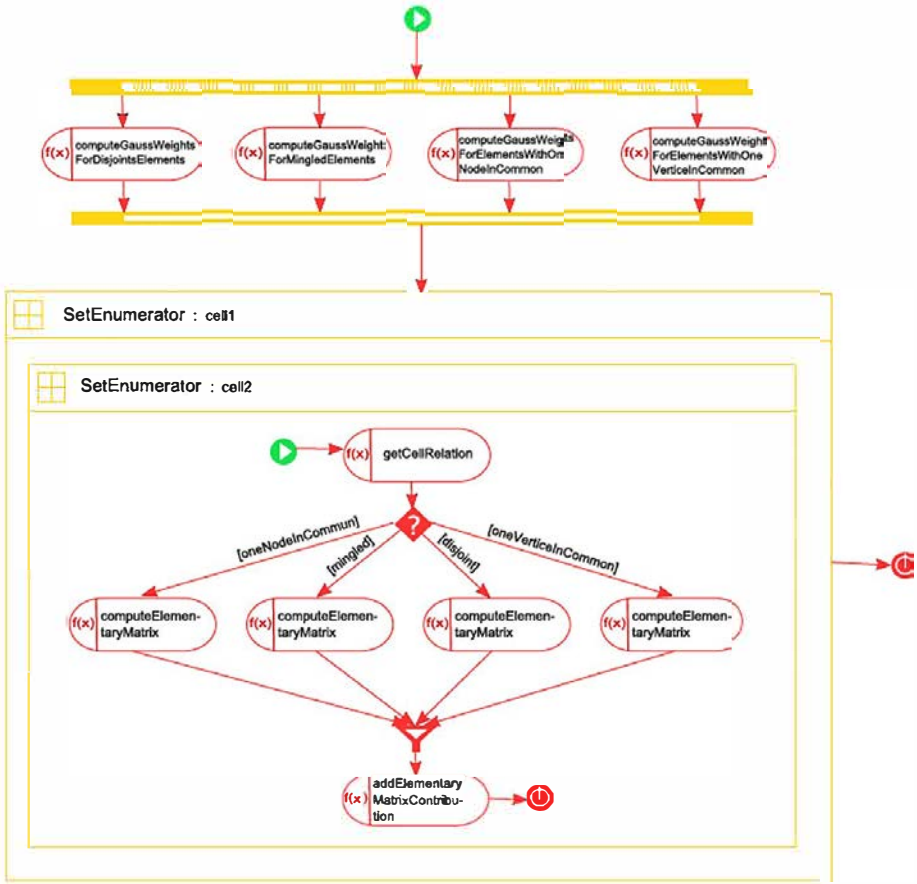
Finally, Fig. 17 contains the last control flow structure that was not presented: conditional connections (`HPCIf` and `HPCEndIf`).

5.3 Benchmarks

The modeled application was used to generate an executable program based on the *Arcane* framework as described in Sect. 4. This section presents the results of benchmarks performed with this program (based on C++ with Arcane) and the original version of the application (based on Fortran with MPI) on the *Tera100* supercomputer—a Bull supercomputer composed of 4370 node servers with 138 368 Intel Xeon cores, an Infiniband interconnect system with a performance of 1,05 petaflops.

First we conducted benchmark to compare the two applications as we wanted to see whether the application produced with the *MDE4HPC* approach was at least as efficient as the application developed in a traditional way. This benchmark was performed on an object with 50,000 meshes. The results, which are presented in Fig. 18, are extremely encouraging. It can be seen that the execution time of the EMPrototype production version and that obtained with *MDE4HPC*, called *EMPrototypeMDE*, are very much similar. Despite being functionally equivalent, the difference in terms of frame-

Fig. 17 `HPCFlowDescriptor` of the *ComputeContributionsMatrix* component



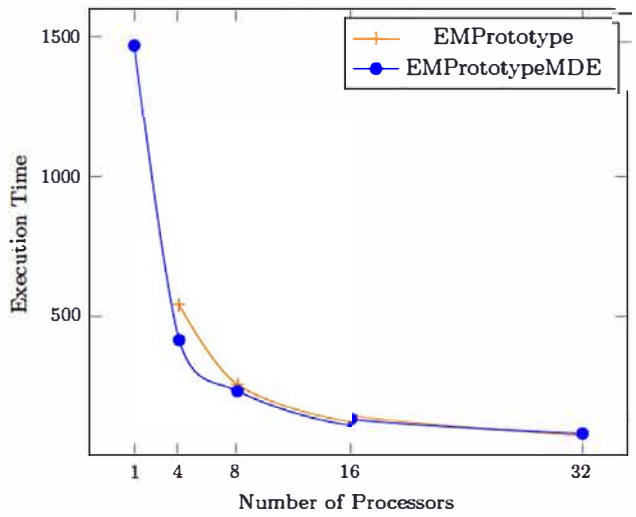


Fig. 18 EMPrototypeMDE : computation time comparisons

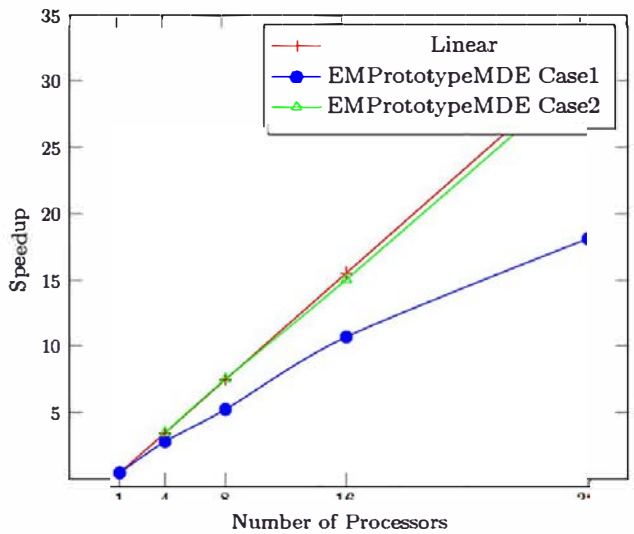


Fig. 19 EMPrototypeMDE : Scalability study

works and languages (before the assembler *[sic]*) explains the slight differences in performance.

Second we conducted another benchmark on *EMPrototypeMDE* alone this time to assess its scalability. This benchmark used two test cases, the first one with an object composed of 10,000 meshes and the second one with an object composed of 50,000 meshes. The results exposed in Fig. 19 show a noteworthy speedup on the first case (Case 1) tending to show the operational nature of the generation of *scalable* code from models. This is all the more true in

so far as this scalability improves as the scale of the problem increases (Case 2). But we must clarify that scalability obtained here is rather due to the algorithm used than the development approach.

5.4 Experiments takeaway and discussion

In this section, we presented a two-stage evaluation of the *MDE4HPC* approach. The criteria mainly addressed in this evaluation are the *scalability* and the *viability* of *MDE4HPC*. However, our approach provides improvements outside of the criteria assessed here. For example, the *HPCML* language is especially useful in *formalizing the specification of simulation codes*. This characteristic is a decisive factor in knowledge capitalization and the transmission of skills.

This case study enabled us to better understand the behavior of the *HPCML* language when modeling a different physical domain and fits in with our previous works where we showed viability in terms of performance of the code generated by the *MDE4HPC* approach as well as a reduction in maintenance costs, especially when porting an application to a new supercomputer [50].

The reduction in terms of maintenance costs was not evaluated again in this new evaluation mainly due to resource constraints. This evaluation is part of our future plans and will be conducted after the deployment of the next generation of supercomputers. Nevertheless, the preliminary results obtained in the different experiments already performed, convinced us that the benefits induced by the use of automation and by the capitalization of the best practices, if generalized to an entire organization, would lead to substantial gains.

The model of the application presented here uses all the concepts we proposed to express parallelism. This model thus provides an adequate basis to run the experimentation needed to define the $HPCML_e$ layer as addressed later when discussing future perspectives.

In order to validate the transformation chains and the code generation, as part of the future work, we propose the *uncertainty quantification*—classical technique in HPC, adapted to the model-based context. It consists in making some parametric studies on the models, because several numerical models can solve on physical one, but also on the optimization of the code generator. Based on these studies, it would be possible to quantify the ratio precision/performance to get the best compromise, depending on the specific needs, in terms of deadline, performance or supercomputer availability constraints.

The performed evaluation allowed us to experience several of the benefits of the use of model-driven development in the context of HPC. In the order of their perceived importance these benefits are:

- MDD enables a selective view of the system and the separation of concerns, through the layered definition used in *MDE4HPC*;
- The use of code generators that lead to a high quality of the obtained code (in terms of performance);
- Allows for domain-specific languages, with a look and feel close to the domain-specific culture to be used instead of raw programming code, thus increasing the accessibility of the domain experts and reducing the cognitive effort required for developing applications;
- Facilitates the apprehension of an HPC application, through offering various abstract views of the system.

Up to this stage, our project did not had the opportunity to take advantage of lots of other promising openings offered by the introduction of models in this context such as reasoning on the abstract model to perform early verification and validation, to investigate non-functional properties of the model, etc. Even so, the results obtained up to this stage make us very eager to investigate further on such openings as part of our future work. The next section complements the quick overview of the model-driven engineering particular benefits that we exploited in our approach.

6 Lessons learned

In the previous sections, we introduced *MDE4HPC*—our approach for adding abstraction in HPC applications by using MDE techniques. In addition to that, a set of domain-specific languages stemming from our method with a focus on $HPCML_n$, the toolset supporting our framework and finally a multi-dimensional validation that assessed the relevance of the current approach have been introduced.

We had the chance of being involved in a full-fledged experiment lasting more than four years, applying MDE principles in high-performance computing. Looking back, we realize that some of the choices we made (with greater or lesser awareness), although seeming reasonable at the time, may be subject to improvement.

In the present section, we analyze the experience we gained and draw a set of lessons from our experimentation of MBSD on a new application domain. Through these lessons learned, we hope to contribute to a wider dissemination of MBSD in a higher number of application domains. To establish these lessons learned, we adopted the perspectives of both the end user of our solution (developer of HPC applications: mainly the numerical analyst presented in Fig. 3) and the developer of MBSD solutions (mainly the software engineer presented in Fig. 3). The current analysis emerges from experience and is in line with other efforts [4,16,30,31,37,42] to capitalize on insights, awareness and best practices obtained from concrete settings.

6.1 The introduction of MBSD should not neglect domain-specific expectations

As we stressed in the introduction, the quest for performance is a central characteristic of high-performance and scientific computing. The goal we targeted in the current experiment was to show that we can obtain using MBSD similar performance to what we can obtain using traditional techniques, with maintainability, portability and customizable code generation as a bonus. Indeed, we demonstrated that the generated code was as effective as the existing code with highly acceptable scalability results (see Sect. 5.3).

Looking back, we realize we should have thought more about HPC practice and put more effort into obtaining a significantly more effective code, which would have earned more interest from simulation code developers. Such performance increases should indeed be feasible, as the rise in the level of abstraction opens new capabilities for optimization (huge deployment of loops, optimum prefetching, generalized inlining, etc.) that are inaccessible when writing code manually.

This lesson seems to us to be something that can be generalized whatever the domain. For example, in real-time and embedded systems, MBSD should show that it adequately satisfies both problems of criticality and existing technical issues. In applications with a strong "Time-To-Market" constraint, rapidity of evolution or production of the code will have to be proved.

6.2 The overall payoff for switching to MBSD is neither immediate nor predictable

Our experiment focused on the use of modeling in order to facilitate the development and maintenance of scientific computing applications. The models obtained in this process could be exploited in directions that are "classical" for the modeling community, but are not used these days by the HPC community. These extras could include the formal verification of some properties on the model, a more precise and better sustained (and possibly formal) characterization of the needs for particular algorithms in terms of platform resources or architectures, an abstract description of timing properties that may lead to model-level time analysis for existing algorithms, etc.

The fact that there is not yet any direct demand for such analysis by the HPC community is due to the current habits of having low-level code that is not readily operable by analysis tools. Adding more abstraction should open the door to a wide range of model-level analyses, most of them yet to be defined in order to carefully address the needs of the HPC community.

6.3 The DSL must comply with usual practices in the domain

Our approach for adding abstraction to scientific computing applications includes the definition of a domain-specific modeling language (cf. Sect. 3). Following current practice in applying MBSD, we used MBSD tools to define a graphical syntax and were proud to end up with a clean, efficient language that complies with expectations. However from the user's perspective, it may have been wiser to have an alternative textual syntax for this language.

One of the priorities for future work is the definition of a textual language for describing low-level algorithms (*HPCAlgorithmicContent*). The syntax of the Liszt language [20] could be a good candidate as its implementation based on Scala makes it possible to use it in ArchiMDE as a generic language for the description of algorithms. Even if such a domain-specific language will necessarily differ from Fortran, it should look familiar to the developer of simulation code.

6.4 MBSD introduces overhead in terms of complexity

MBSD tools often introduce (too much?) complexity, with respect to the existing technological levels. The technological break (moving on from Fortran to Eclipse Modeling Framework, or from the Nedit to Eclipse) is often extremely significant. The considerable personal investment for future users may be dissuasive. The overall cost of setting up the technologies and investments (in operational terms) is significant for a remote (and not understood by everybody) return on investment.

Computer science and engineering curricula should allow students to acquire modern software technologies. In parallel with this, the MBSD community must devote efforts to make using their tools simpler and more intuitive. There is a large consensus (see for instance [15]) that efforts are needed in making modeling tools more accessible to a wider range of users. Our experience confirms these needs.

6.5 MBSD can still make progress

According to the feedback we received, the use of modeling techniques suffers from two major weaknesses that are linked: co-evolution of models and team work.

During the definition of the *HPCML* metamodel and during the actual language use on the evaluations, we frequently faced the need to make changes which led to painstaking model migration activities. Even though the number changes in the metamodel and language definition started to fall once a certain maturity point was reached, changes must still be expected for newer versions. To aim at an industrial use of its technologies, MBSD needs a flexible and effective model

version management mechanism handling both model and metamodel versioning. We are aware of interesting results in this area, but the fact that commonly used tools lack effective version management makes their usage difficult to defend in an industrial setting. We therefore feel that there is an urgent need for standardization and integration of such mechanisms in environments like Eclipse.

7 Related work

There is a rich diversity of parallel programming solutions available for developing scientific software. This situation stems from the complexity and the fast evolution of the hardware we described in Sect. 1. The existing solutions tackle the development of scientific applications at different levels of abstraction. The current section gives an overview of the capabilities of existing solutions for developing parallel numerical simulation at various levels of abstraction, in order to understand how they complement and differ from our approach. The overview covers both solutions consisting in adding abstraction at the programming language or environment level, as well as a few solutions that are sensitive to model-driven techniques.

7.1 Mainstream programming solutions

For more than half a century, the Fortran language has been the reference in HPC. Even though C++, Python or even Java are gaining interest in the scientific community, Fortran 90 is still massively used in scientific computing. However, none of these languages are capable of exploiting massively parallel architectures directly, and they must rely on external components. The most widely used of these complementary solutions is certainly MPI (Message Passing Interface) [56] which is a standardized message-passing system. It is mainly used to exploit distributed memory architectures. For shared memory architectures in addition to the usual threading libraries, the OpenMP (Open Multi-Processing) [19] which is an API that supports multi-platform shared memory multiprocessing is quite popular. Even though they are widespread, these solutions suffer from the problems we highlighted in the introduction (see Sect. 1.1).

The use of libraries that contain optimized math routines for science such as the Intel Math Kernel Library (MKL) [1] is a common practice. On the bright side, these libraries hide most of the complexity of the underlying hardware, and they are usually developed by experts who write very efficient code. However, they can only be used to parallelize the portion of the code where they take part. Furthermore they require the use of predefined data structures and thus can demand costly data conversions mainly if a developer want to mix several libraries of this kind.

7.2 PGAS model

A programming model that gained a certain popularity in recent years is the Partitioned Global Address Space (PGAS). This model offers a shared address space model that simplifies programming while exposing data/thread locality with the aim of enhancing both productivity and performance. PGAS programming solutions include Coarray Fortran [53] and Unified Parallel C (UPC) [59]. Coarray Fortran is a small Fortran extension adding an explicit notation for data decomposition, with the aim of improving its native support for parallelism. Similarly, UPC is a PGAS extension of the C programming language designed for high- performance computing on large-scale parallel machines[12].

The PGAS model relies on a static allocation approach. The APGAS (Asynchronous Partitioned Global Address Space) model extends and solve some of the problems of the PGAS model. APGAS programming solutions were developed as part of the High Productivity Computing Systems (HPCS) program launched in 2002 by the DARPA [60]. Three novel programming languages emerged: Chapel [13] (Cray), Fortress [3] (SUN) and X10 [14] (IBM). The principal drawback of this approach is the need to develop high-performance compiler, debugger and implementation for each existing architecture.

7.3 Dynamic scheduling

New hardware architectures cover various aspects (such as fault tolerance, energy consumption management while offering heterogeneous execution units,…) that make it difficult to dynamically schedule the calculus. To answer this problem, projects such as StarPU [6] and Qilin [39] describe applications in terms of tasks and let an external scheduler take care of the resource allocation, based on run-time information. These approaches that suffer from a very high dependency on the external scheduler lead to unpredictable execution performances, and hardware evolution may require significant rethinking of the application.

7.4 Component assembly

The *Common Component Architecture (CCA)* [2] is a component model specification targeting scientific computing. CCA has the merit of allowing a combination within the same system of components that use various programming languages.

Other approaches, such as the *High-Level Component Model (HLCM)* [8], also based on component-based architectures, serve to specify the composition by adding custom composition operators, without touching the components themselves. The abstract HLCM models are specialized

using model-based approaches, such as those done for CCA and Corba [26].
While the component-based developments proved successful, the existing approaches are not constraining the internal component architecture enough, in particular with respect to handling the parallelism, and thus offer limited portability of the components on various hardware platforms.

7.5 Legacy code improvement

Macro-based approaches such as HMPP (Hybrid Multi-core Parallel Programming environment) [9] offer a respectable solution for improving legacy code. However, as their use is based on compiler directives which limit the separation of concerns, this solution has limited interest for new developments.

7.6 High-level programming language-based solutions

One strategy to deal with the complexity of general-purpose GPU is to use high-level programming languages, such as OCaml, and a library to handle GPGPU programs and data transfers between devices. This is the way taken by SPOC [10]—a library that expresses GPGPU kernel through interoperability with common low-level extensions. While promising, this approach is currently limited in terms of handled data type and it offers another attempt to provide a higher level of abstraction in the GPGPU programming.

7.7 Model-based approaches

The use of model-driven techniques in the context of high-performance computing is rather marginal, but some attempts at using MBSD in HPC and scientific computing do exist.

Trying to answer the acute need of support for maintenance existing in HPC applications, ForUML [45] proposes a software tool to extract UML class diagrams from Fortran code. This reverse engineering tool seems to facilitate maintenance and refactoring.

In [5], the authors propose a model-driven approach to analyze, model and select feasible mappings of parallel algorithms to a parallel computing platform. The approach claims to offer support for an early analysis of potential mappings with respect to speedup and efficiency and generate the platform-specific model and the source code.

The GASPARD [23] design framework addresses massively parallel embedded systems at a higher level of abstraction. The approach relies on a repetitive Model of Computation (MoC), which offers a powerful and factorized representation of parallelism in both system functionality and architecture. Embedded systems are designed using the MARTE (Modeling and Analysis of Real-time and Embedded systems). Using MBSD paradigm, MARTE models are refined toward lower abstraction levels, which make the design space exploration possible. For the moment, the approach transformation chains the transformation chain toward SystemC that only manages software execution on a processor-based architecture at the PVT (Programmer View Timed) abstraction level.

Another project tackling our problematic by raising the level of abstraction is Liszt [20]. It aims at solving partial-differential equations on meshes by providing a domain-specific language built on top of Scala. Due to its qualities and common philosophy with *HPCML* , we plan to add Liszt as an additional target of the transformation process in the ArchiMDE tool. This would demonstrate the capability of *HPCML* to target multiple high-level platforms (Arcane, Liszt). Singe [7] is a DSL compiler for a dedicated physics simulation problem that leverages warp specialization to produce high-performance code for GPUs. Singe demonstrates that the generated code can be much faster than previously optimized data-parallel GPU kernels.

A more complete view of related work can be found in [51]. As far as we know, none of these approaches target a fully fledged solution for introducing abstraction in scientific software applications though the use of models. The *MDE4HPC* approach presented in this paper was previously introduced [51] and evaluated [50]. Parts of the *HPCML* language have been also introduced [52]. This paper extends these publications by giving more details about the approach (Sect. 2) and the language (Sect. 3), by providing an additional evaluation regarding performance (Sect. 5) and by summarizing the lessons learned from the overall project (Sect. 6).

8 Conclusions

This paper reports on a 4-year experiment in raising the abstraction level in the development of scientific applications using high-performance computing through applying model-driven software development (MBSD) techniques. High-performance computing is not a field where MBSD is traditionally used. A recent search we have performed shows that none of the SoSym papers addresses this field, and among the papers submitted in 2014 at the MODELS conference,[3] the premier venue conference for modeling techniques, only one paper (0.66 %) addressed this topic. We had the unique opportunity of being involved in an industrial project that aimed at adding more abstraction in high-performance scientific applications through the use of modeling. The duration of this study as well as the feedback we gathered through informal interviews from approximately 10 domain experts makes this experiment more than just yet

[3] http://www.modelsconference.org/.

another standard application of MBSD principles and allows us to report on some lessons learned that could be useful to other experiments applying modeling techniques in novel fields.

We have presented *MDE4HPC*, our approach for applying MBSD in the development of scientific simulation software. This approach is based on the definition of a multi-layered domain-specific language *HPCML*. This language has a modular and layered definition that serves to capture information specific to the modeling of the numerical aspects, of the physical model and of the computation information. For the definition of this language, we defined its metamodel as well as a graphical syntax addressed to domain experts, thus making the manipulation of models transparent to this class of users, in general without formal training in computer science. In order to make our approach effectively usable, we made the required tool developments to integrate it in existing development tool chains, as described in Sect. 4. We have reported on the evaluation that we conducted on applying our approach to realistic simulation problems as well as on the experience gained and lessons learned.

Given confidentiality issues related to the project, there is no public Web site presenting it or the tools developed in this context. Most of the artefacts we developed in this project (metamodels, model transformations, language definitions, etc.) are, however, unclassified and are available on request to one of the authors of this paper.

The pragmatic approach that we adopted consisted in grounding our entire toolset on an already existing tool ecosystem developed in-house—Arcane [25], as well as some optimizations of the Fortran compiler. This led us to a multi-layered approach with layers corresponding to the operating system, to the compiler, to the Middleware (Arcane) and to the abstract model corresponding to our *MDE4HPC* approach. This layered structuring, detailed in Sect. 4, has the advantage of reusing already available and operational components, where optimized code generators from intermediate formats are already available. The current study shows that it is possible to have an automatized transformation chain, allowing to use the models as core artefacts for reasoning, while generating customized code targeted to the architecture we focus on.

The most generally applicable outcomes of our study are synthesized in the section dedicated to the lessons learned (Sect. 6). A basic summary of these lessons would be the following:

> in spite of the undeniable advantages in terms of maintenance facility, separation of concerns, etc., being capable of just reproducing the technical results obtained in a solution not using modeling and abstraction is not enough, to allow the deployment and the adoption of such solutions.

At the beginning of this study, we started from the hypothesis that the rise of the level of abstraction introduced by the use of models, and the openings for the separation of concerns, maintenance, etc., would be enough to allow its adoption. It would be interesting to have more empirical studies related to the resistance to change related to the introduction of new technologies in particular in the field of scientific applications.

The current work opens the way to more exciting studies. One direction would be to enrich the current model with non-functional information related, for instance, to uncertainty characterizations of some calculus, performance analysis of some algorithms, resource consumption information (in terms of time, energy or hardware charge), etc. This non-functional information expressed at the model level would open the way to parameterizations following various criteria, possibly defined dynamically that would lead to more flexible computations. Depending on certain resource availability parameters, the execution would vary dynamically to become either more trustworthy (thus minimizing the uncertainty parameter), more rapid (minimizing the execution times) or less energy consuming. Interesting openings exist in this context with work done recently in the context of Models@runtime [11].

This study suggests the need for an evolution in terms of services offered by hardware providers. Today, they provide compilers optimized for the hardware architecture that they are building. In the future, we feel this should evolve toward providing, as well as optimized compilers, (parametrized) optimized code generators. This would naturally demand a larger adoption of a language such as the one we introduce in this paper *HPCML*, maybe through some standardization attempts such as in the case of OpenCL [34,58].

Another direction that this work could be taken concerns collaborative and co-evolutive modeling. One of the motivations of this work was to enable the separation of concerns during the development of scientific applications. We achieved this through the definition of a multi-layered DSL. The various domain experts involved in the development of scientific applications interact with various parts of the DSL, and thus, the integration of the various aspects is achieved by means of composition by construction. An alternative approach, worth being investigated, would consist in defining DSLs for the various domain experts, so that each language can live its own life (that would be useful in terms of maintenance and to alleviate tool support). However, this would require some mechanisms for composing the various DSLs and could benefit from existing work such as GEMOC.[4]

An interesting opening offered by this study would benefit the modeling community. Indeed, one of the areas of current interest concerns the scalability of the modeling techniques to

[4] http://gemoc.org/.

big models from large-scale systems. The high-performance computing field does have some experience in handling large-size data. We could take advantage of the techniques currently used in HPC and apply them to more effectively handle large-scale models, thus leading to new research in our domain that has still to be invented, a kind of High-Performance Modeling.

Last but not least, introducing the use of models in a field such as HPC would open the way to apply other model analysis tools specific to MBSD. While in this project we only targeted code generation, we can naturally envisage applying verification and validation techniques, or tailoring the existing verification approaches to the specificity of this field. One may think that the classical space explosion problem facing model-checking tools would make them inapplicable in this field. However, in several HPC applications the need for large computing capabilities is due to the size of data rather than to the complexity of the algorithms. Here, raising the level of abstraction and applying verification techniques at the algorithms level may allow early detection of some errors before actually deploying algorithms on concrete hardware platforms or applying them to big data.

References

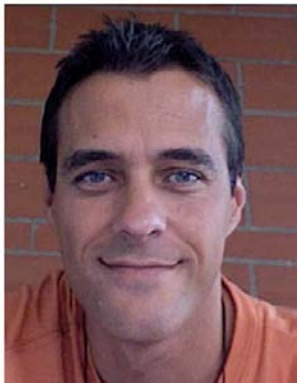
1. Intel math kernel library (intel mkl) v11.0. http://software.intel.com/en-us/intel-mkl/
2. Allan, B.A., Armstrong, R., Bernholdt, D.E., Bertrand, F., Chiu, K., Dahlgren, T.L., Damevski, K., Elwasif, W.R., Epperly, T.G.W., Govindaraju, M., Katz, D.S., Kohl, J.A., Krishnan, M., Kumfert, G., Larson, J.W., Lefantzi, S., Lewis, M.J., Malony, A.D., Mclnnes, L.C., Nieplocha, J., Norris, B., Parker, S.G., Ray, J., Shende, S., Windus, T.L., Zhou, S.: A component architecture for high-performance scientific computing. Int. J. High Perform. Comput. Appl. **20**(2), 163–202 (2006)
3. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G. L. Jr., Tobin-Hochstadt, S.: The Fortress Language Specification. Technical report, Sun Microsystems Inc, (2007). http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf
4. Aranda, J., Damian, D., Borici, A.: Transition to model-driven engineering—What is revolutionary, what remains the same? In: 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012, pages 692–708 (2012)
5. Arkin, E., Tekinerdogan, B., Imre, K.M.: Model-driven approach for supporting the mapping of parallel algorithms to parallel computing platforms. In: MoDELS, pages 757–773, (2013)
6. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
7. Bauer, M., Treichler, S., Aiken, A.: Singe: Leveraging warp specialization for high performance on gpus. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pages 119–130, New York, NY, USA. ACM (2014)
8. Bigot, J., Pérez, C.: High performance composition operators in component models. In: Foster, I., Grandinetti, L., Joubert, G.R., Gentzsch, W. (eds.) High Performance Computing: From Grids and Clouds to Exascale, Volume 20 of Advances in Parallel Computing, pp. 182–201. Amsterdam, IOS Press (2011)
9. Bodin, F.: Keynote: compilers in the manycore era. In: HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, pp. 2–3, Berlin, Heidelberg. Springer (2009)
10. Bourgoin, M., Chailloux, E., Lamotte, J.L.: Efficient abstractions for GPGPU programming. Int. J. Parallel Program. **42**(4), 583–600 (2014)
11. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living models-ten principles for change-driven software engineering. Int. J. Softw. Inform. **5**(1–2), 267–290 (2011)
12. Cantonnet, F., Yao, Y., Zahran, M.M., El-Ghazawi, T.A.: Productivity analysis of the UPC language. In: IPDPS (2004)
13. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
14. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '05, pp. 519–538. ACM (2005)
15. Clark, T., France, R.B., Gogolla, M., Selic, B.V.: Metamodeling model-based engineering tools (Dagstuhl Seminar 13182). Dagstuhl Rep. **3**(4), 188–226 (2013)
16. Clark, T., Muller, P.: Exploiting model driven technology: a tale of two startups. Softw. Syst. Model. **11**(4), 481–493 (2012)
17. Clayberg, E., Rubel, D.: Eclipse: Building Commercial-Quality Plug-ins (2nd Edn) (Eclipse). Addison-Wesley Professional, Boston (2006)
18. Colton, D.L., Kress, R.: Integral Equation Methods in Scattering Theory. Wiley, New York (1983)
19. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. Comput. Sci. Eng. **5**, 46–55 (1998)
20. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: a domain specific language for building portable mesh-based pde solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 9:1–9:12. ACM (2011)
21. Dongarra, J.J.: Performance of various computers using standard linear equations software. SIGARCH Comput. Archit. News **20**(3), 22–44 (2014)
22. Foxy, G., Hawick, K., White, A.: Characteristics of hpc scientific and engineering applications. In: Second Pasadena Workshop on System Software on Tools for High Performance Computing Environments (1996)
23. Gamatié, A., Le Beux, S., Piel, E., Ben Atitallah, R., Etien, A., Marquet, P., Dekeyser, J.-L.: A model-driven design framework for massively parallel embedded systems. ACM Trans. Embed. Comput. Syst **10**(4), 39:1–39:36 (2011)
24. Gonnord, J., Leca, P., Robin, F.: Au delà de 50 mille milliards d'opérations par seconde! La Recherche **393**, 39–44 (2006)
25. Grospellier, G., Lelandais, B.: The Arcane development framework. In: POOSC '09. ACM (2009)
26. Group, O.M.: Corba component model 4.0 specification. Technical report, Object Management Group (2006)
27. Group, O.M.: MOF 2.4 Specification. Technical report (2011). http://www.omg.org/spec/MOF/2.4/
28. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with the Google Web Toolkit. Manning Publications Co., Greenwich (2007)
29. Harrington, R.: Time-harmonic electromagnetic fields. In: IEEE Press Series on Electromagnetic Wave Theory. Wiley (2001)

30. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE, pp. 471–480 (2011)
31. Kalliamvakou, E., Palyart, M., Murphy, G., Damian, D.: A field study of modellers at work. In: 2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering (MiSE) (2015)
32. Kepner, J.: HPC productivity: an overarching view. Int. J. High Perform. Comput. Appl. **18**(4), 393–397 (2004)
33. Keutzer, K., Massingill, B.L., Mattson, T.G., Sanders, B.A.: A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In: Proceedings of the 2010 Workshop on Parallel Programming Patterns, ParaPLoP '10. ACM (2010)
34. KhronosGroup. The OpenCL specification v1.2. Technical report (2011). http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
35. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. Springer, New York (1997)
36. Kirk, D.: NVIDIA CUDA software and GPU parallel computing architecture. In: ISMM, pp. 103–104 (2007)
37. Kuhn, A., Murphy, G.C., Thompson, C.A.: An exploratory study of forces and frictions affecting large-scale model-driven development. In: 15th International Conference on Model Driven Engineering Languages and Systems, MODELS, pp. 352–367 (2012)
38. Larus, J.R.: Spending Moore's dividend. Commun. ACM **52**(5), 62–69 (2009)
39. Luk, C.-K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 45–55. ACM (2009)
40. Merali, Z.: Computational science:...error. Nature **467**, 775–777 (2010)
41. Miller, J., Mukerji, J.: Mda guide version 1.0.1. omg/2003-06-01. Technical report, OMG, (2003)
42. Mohagheghi, P., Dehlen, V.: Where is the proof? - a review of experiences from applying mde in industry. In: Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications, ECMDA-FA '08, (2008)
43. Moody, D.L.: The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Software Eng. **35**, 756–779 (2009)
44. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8), 114–117 (1965)
45. Nanthaamornphong, A., Morris, K., Filippone, S.: Extracting uml class diagrams from object-oriented fortran: Foruml. In: Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, SE-HPCCSE '13, pages 9–16, New York, NY, USA. ACM (2013)
46. Nassiet, D., Livet, Y., Palyart, M., Lugato, D.: Paprika: rapid UI development of scientific dataset editors for high -performance computing. In: Proceedings of the 15th international conference on Integrating System and Software Modeling, SDL'11, pages 69–78. Springer-Verlag, (2011)
47. Northover, S., Wilson, M.: SWT: the standard widget toolkit. Addison-Wesley, The Eclipse series (2004)
48. Oudin, H.: Méthode des éléments finis. http://cel.archives-ouvertes.fr/cel-00341772/PDF/bouquin.pdf, (Sept. 2008)
49. Palyart, M.: Une approche basée sur les modèles pour le développement d'applications de simulation numérique haute-performance. PhD thesis, Université Paul Sabatier - Toulouse III, (2012)
50. Palyart, M., Lugato, D., Ober, I., Bruel, J.-M.: Improving scalability and maintenance of software for high-performance scientific computing by combining MDE and frameworks. In: Proceedings of the 14th international conference on Model driven engineering languages and systems, MODELS'11, pages 213–227. Springer-Verlag, (2011)
51. Palyart, M., Lugato, D., Ober, I., Bruel, J.-M.: MDE4HPC: An approach for using Model-Driven Engineering in High-Performance Computing. In: 15th System Design Languages Forum (SDL 2011), (2011)
52. Palyart, M., Lugato, D., Ober, I., Bruel, J.-M.: HPCML: A Modeling Language Dedicated to High-Performance Scientific. In: 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing (MDHPCL). ACM, (October 2012)
53. Reid, J.: Coarrays in the next fortran standard. SIGPLAN Fortran Forum, 29(2), 2010
54. Shan, H., Singh, J.P.: A Comparison of MPI, SHMEM and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessors. Int. J. Parallel Program. **29**(3), 283–318 (2001)
55. Shankland, S.: In newest tally, supercomputing progress tapers off. CNET, november, (2014). http://www.cnet.com/news/in-newest-tally-supercomputing-progress-tapers-off/
56. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The complete reference. MIT Press, Cambridge, MA (1996)
57. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional, (2009)
58. Stone, J., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering **12**(3), 66 (2010)
59. UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, (2005). http://www.gwu.edu/upc/publications/LBNL-59208.pdf
60. Weiland, M.: Chapel, Fortress and X10: Novel Languages for HPC. Technical report, The University of Edinburgh, (October 2007). http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0706.pdf

Ileana Ober is associate professor at the University of Toulouse and researcher at IRIT, France. She holds a PhD from the Polytechnic Institute of Toulouse in 2001 and a Habilitation from the University of Toulouse since 2010. She was research engineer at Telelogic, where she worked on several research projects and was actively involved in the UML action semantis definition at the OMG and with the definition of SDL 2000 at ITU. After a postdoc at VERIMAG studying model-driven validation of real-time systems using UML timing constraints in the context of the 1ST project OMEGA, she joined IRIT as associate professor, where she works on developing techniques to include domain-specific language-based development in a model driven engineering framework and on applying modeling techniques on high performance computing. She co-chaired MODELS 2008 and SDL2011, she initiated the Modeling Wizards International Master Class on MDE and she organized several workshops at MODELS. She regularly serves as expert for H2020 project proposal evaluations.

Marc Palyart is a postdoctoral researcher in the Software Practices Lab from the University of British Columbia. He received his PhD and MSc in Computer Science from the University of Toulouse. He also holds a BSc (Hons) from the Dundalk Institute of Technology. His research interests include software engineering and software evolution.

David Lugato is a research engineer at the Commisariat à l'Energie Atomique (CEA). He obtained his Habilitation (HDR) from the University of Bordeaux in 2015. He developed the AGATHA software (Automatic test generation with symbolic execution) at CEA Saclay from 2000 to 2004. To tackle the combinatorial explosion limitations, the spectrum of research was broad and developments required the assembly of complex and disparate software components : rewriting machine, constraint solver and graph theory. From 2004 to 2012, he has lead a project of an operations research simulation software. The development of this complex software due to its size and the multitude of modeled physical domains requires raising the level of abstraction to ensure greater sustainability and facilitate the accessibility to physicists. Since 2012, he is software architect of a large simulation software for atmospheric reentry with special focus on solutions for coupling multi physics and multi-scale problems and study of a DSL definition for future exascale supercomputers.



Jean-Michel Bruel received his Ph.D. from the University Paul Sabatier (Toulouse) in December 1996. From September 1997 to August 2008, he was associate professor at the University of Pau and Member of the LIUPPA (Laboratoire d'Informatique de l'Universitè de Pau et des Pays de l'Adour) from 2000 to 2008. He has defended his "Habilitation à Diriger des Recherches" in December 2006 and is since 2008 full professor at the University of Toulouse. He has been head of the Computer Science department of the Technical Institute of Blagnac from 2009 to 2012. Currently, he is the head of theMACAOteam (Models, Architectures, Components, Agility and prOcesses) of the IRIT (Institut de Recherche en Informatique de Toulouse) CNRS laboratory. His research areas include Model-Based System Engineering (MBSE) and more precisely formal requirements, model and method integration.