THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par *Marc PALYART-LAMARCHE*
Le *18 décembre 2012*

Titre : *Une approche basée sur les modèles pour le développement d'applications de simulation numérique haute-performance*

JURY
Robert FRANCE – Professeur, Colorado State University, États-Unis (rapporteur)
Raymond NAMYST - Professeur, Université Bordeaux 1 (rapporteur)
Jean-Marc PIERSON – Professeur, Université Toulouse 3 (président)
Benoit BAUDRY – Chargé de recherche HDR, INRIA Rennes (membre)
Bran SELIC - Malina Software Corp, Canada (membre)
Jean-Michel BRUEL – Professeur, Université Toulouse 2 (directeur de thèse)
Ileana OBER - Maître de conférences HDR, Université Toulouse 3 (co-directrice de thèse)
David LUGATO – Ingénieur chercheur, CEA/CESTA (co-directeur de thèse)

Ecole doctorale : *Mathématiques Informatique Télécommunications (MITT)*
Unité de recherche : *Institut de Recherche en Informatique de Toulouse (IRIT, UMR 5505) / Commissariat à l'Énergie Atomique et aux Énergies Alternatives - Centre d'Études Scientifiques et Techniques d'Aquitaine (CEA/CESTA)*
Directeur(s) de Thèse : *Jean-Michel BRUEL, Ileana OBER et David LUGATO*
Rapporteurs : *Robert FRANCE et Raymond NAMYST*

Remerciements

La position initiale de ces remerciements ne doit pas fourvoyer le lecteur profane. Cette section est, en effet, la clôture d'une aventure humaine de longue haleine et a fait l'objet de ma part d'une convoitise effrénée tout au long de la rédaction de ce manuscrit. Voici donc enfin arrivé le moment de remercier toutes les personnes qui ont participé de près ou de loin à cette aventure et qui ont permis qu'elle aboutisse.

Je tiens tout d'abord à remercier mon encadrant au *CEA* David Lugato pour le temps considérable qu'il m'a consacré au cours de ces trois dernières années et pour le savoir qu'il a su me transmettre. Je remercie également mes directeurs de thèse Jean-Michel Bruel et Ileana Ober pour leur soutien et leurs conseils.

Je souhaite remercier l'ensemble des membres du jury à qui je porte toute mon admiration et dont je suis fier qu'il soit associé à cette thèse. À commencer par Robert France et Raymond Namyst pour avoir accepté de rapporter cette thèse et m'avoir prodigué de précieux conseils. Suivis de Benoit Baudry, Jean-Marc Pierson et Bran Selic pour avoir accepter de prendre part à ma soutenance de thèse.

Passons maintenant aux nombreuses personnes du *CEA*. En premiers lieux, je tiens à remercier tous les membres de *LEC* pour leur bonne humeur et leur soutien : Sébastien pour sa version des « mathématiques appliquées pour les nuls » ainsi que pour les heures passées à lire du *Fortran* en ma compagnie ; Sébastianos qui a surement passé autant de temps à relire ce manuscrit que j'en ai passé à l'écrire ; Didier pour les 108 discussions que nous avons eu sur *EMF* ; Yohan qui a toujours été là pour me remonter le moral en me montrant par l'exemple que l'on peut quand même s'en sortir lorsque le sort s'acharne contre vous ; Jean-Philippe qui a vécu la difficile expérience de partager le bocal avec moi ; Thierry mon fournisseur de maillages ; Fabien le développeur de *VisIt* qui a toujours été là pour répondre à mes questions sur la visualisation scientifique ; Paul notre *Wikipedia* à nous en version interactive ; Philippe pour nos longues discussions ; Christine pour avoir veillé à ce que je reste dans le droit chemin ; Marc mon double et finalement Stéphane et François les maitres incontestés de la contrepèterie et du calembour qui nous ont quitté trop tôt (pour aller dans un autre laboratoire). Je remercie également Nicolas Koeke et Xavier Carlotti de m'avoir accueilli au sein de leur unité respective ainsi que les différentes secrétaires qui se sont succédées : Rosana, Axelle, Anne Pascale, Christine, Martine. Je suis très reconnaissant envers Gilles pour les versions personnalisées d'Arcane accompagnées d'une hotline. Je suis aussi sincèrement reconnaissant envers Murielle et Agnès pour m'avoir guidé dans les

méandres de l'électromagnétisme ainsi qu'à Marc, mon second double, pour nos discussions sur les incertitudes de la vie et des codes de simulation. Et puis, il y a toutes ces personnes que je remercie car elles m'ont permis de décompresser afin de garder les idées claires dans les moments difficiles. Je veux bien entendu parler des membres de la section Vovinam Viet Vo Dao : Xavier, Patrice, Jacques, les deux Jérôme, Cécile, Laure et Jonathan ainsi que des membres de la section ski : Regis, Marielle, Fabien et les autres.

Il est vrai que la distance nous séparant a réduit le nombre de nos rencontres mais je tiens à remercier l'ensemble des membres de l'équipe MACAO de l'IRIT qui m'ont toujours accueilli chaleureusement lors de ces trop rares rencontres. Je remercie également Xavier, J.R. et Laurent du LaBRI pour leurs conseils et leurs vannes toujours adéquates (comme Sheila).

Comme dans toute aventure, il y a des personnes qui vous soutiennent et d'autres dont on aurait préféré ne jamais croiser le chemin, au sens propre comme au figuré dans le cas présent. C'est pourquoi je tiens officiellement à ne pas remercier l'automobiliste qui, par une après-midi de novembre 2010, a décidé de jouer aux auto-tamponneuses avec moi alors que je roulais tranquillement à vélo.

Finalement, je remercie l'ensemble de ma famille dont mes deux sœurs de pas avoir trop honte de leur frère, ma nièce Lola à qui je ne cesse de répéter que les bases sont importantes et puis mes parents qui m'ont permis de poursuivre mes études pour en arriver là. Bien sûr, il ne m'est pas concevable de conclure ces remerciements sans en adresser à Floriane qui malgré le poids de cette thèse sur notre vie de couple a toujours été près de moi et s'est démenée pour que le pot de la soutenance soit un moment inoubliable.

Un modèle de ce manuscrit[1]

1. Dates des *commits* effectués pour la rédaction de ce manuscrit (jour de l'année en abscisse et heure en ordonnée)

Table des matières

Chapitre

1

Introduction

Ainsi s'écoule toute la vie.
On cherche le repos en combattant quelques obstacles ;
et si on les a surmontés, le repos devient insupportable.

Blaise Pascal.

La simulation numérique est un puissant outil permettant de prévoir ou comprendre des phénomènes réels en utilisant des modèles physiques et mathématiques. Elle fut d'ailleurs l'une des principales raisons de l'essor de l'informatique au milieu du XX^{e} siècle.

Dans les années 1960, les développements de codes de simulation numérique impliquaient le plus souvent un seul domaine physique, un seul schéma numérique, et reposaient sur un écosystème logiciel minimal contenant essentiellement un langage de programmation : *Fortran*. En ce début des années 2010, la situation a radicalement changé. On assiste à des développements impliquant plusieurs domaines de la physique, plusieurs schémas numériques et, par dessus tout, une complexification de l'écosystème logiciel nécessaire à ces développements.

La complexité de ces développements logiciels est devenue extrême pour deux raisons. Premièrement, la complexité nécessaire de ces développements, c'est-à-dire celle qui est intrinsèque au problème, a naturellement augmenté au fur et à mesure des avancées scientifiques. Deuxièmement, la complexité accidentelle provenant des méthodes et outils a, quant à elle, explosé à cause de la diversité et du caractère parfois éphémère des solutions logicielles et matérielles disponibles.

Problématique. Trois problèmes majeurs expliquent l'augmentation de cette complexité accidentelle.

Premièrement, les solutions logicielles sont fortement couplées avec les architectures matérielles. Or, face à la demande constante de puissance de calcul, il faut, d'une part, fréquemment remplacer les calculateurs, et d'autre part, introduire des ruptures technologiques matérielles lors de leur renouvellement. Les différences importantes d'architecture lors du changement d'une machine nécessitent de modifier en profondeur les pratiques de programmation et impactent donc lourdement les opérations de migrations logicielles.

Deuxièmement, les solutions logicielles actuelles provoquent un mélange des préoccupations dans le code source : les informations concernant la phy-

sique et les mathématiques sont, par exemple, complètement entremêlées d'informations pour la gestion du parallélisme ou la validation des résultats.

Troisièmement, les solutions de programmation parallèle permettant d'exploiter le maximum de la puissance de calcul d'une architecture matérielle à un instant donné sont, la plupart du temps, volatiles et complexes. Ces langages et leur évolution induisent des couts de formation des développeurs importants. À noter que ces développeurs sont le plus souvent des numériciens pour qui l'informatique ne fait pas partie de leur cœur de métier mais qui est néanmoins une tâche chronophage.

L'ingénierie dirigée par les modèles (*IDM*) [90] est une approche pour le développement logiciel proposant de créer des représentations abstraites : les modèles, qui s'intéressent à la description d'aspects particuliers du système étudié (le logiciel de simulation dans notre cas). L'ensemble de ces modèles est utilisé pour générer, via des transformations de modèles, une partie ou l'ensemble du code source du système modélisé. Notre travail s'inscrit dans cette mouvance en appliquant ces principes au développement d'applications de simulation numérique.

Contributions Les contributions proposées dans cette thèse poursuivent donc un même objectif : celui de faciliter le développement et la maintenance d'applications de simulation numérique haute-performance. C'est dans cette optique que nous avons défini les deux éléments suivants :

- *MDE4HPC*. Une approche définissant comment appliquer les principes généraux de l'ingénierie dirigée par les modèles dans le cas particulier du développement et de la maintenance d'applications de simulation numérique haute-performance.
- *HPCML*. Un langage de modélisation dédié (*DSML*) se trouvant au cœur de l'approche *MDE4HPC*. Ce langage fournit un formalisme pour la description des modèles qui serviront d'entrées au processus de génération.

L'ensemble des travaux contenus dans cette thèse ont été réalisé au Centre d'Etudes Scientifiques et Techniques d'Aquitaine (CESTA) du *CEA/DAM*. Ces travaux s'inscrivent dans le cadre du programme *Simulation* et plus particulièrement du programme *Tera* (cf. section 2.1.2). Nous avons donc eu la chance de pouvoir travailler sur la machine *Tera100*, qui était le supercalculateur le plus puissant d'Europe au cours de cette thèse.

Cette thèse comprend trois parties principales : la présentation du contexte, la description de l'approche basée sur l'*IDM* que nous proposons pour faciliter le développement d'applications de simulation numérique et finalement l'évaluation de cette dernière. Ces trois parties sont suivies par une partie finale qui traite les perspectives et la conclusion.

La première partie (contexte) présente le contexte et la problématique de cette thèse. Dans un premier temps, le chapitre 2 aborde l'univers de la simulation numérique et les

problèmes liés au développement d'applications de simulation numérique. Dans un second temps, ce chapitre introduit les architectures matérielles des supercalculateurs et leur évolution afin de comprendre le lien fort existant entre logiciel et matériel dans ce domaine. Dans le chapitre 3, nous présentons les solutions logicielles disponibles pour le développement d'applications de simulation numérique et nous identifions dans quelles mesures elles répondent aux problèmes introduits dans le chapitre 2. Ce chapitre aborde ensuite la relation entre calcul scientifique et génie logiciel.

La deuxième partie (contribution) est consacrée à la description de notre proposition pour adresser les problèmes rencontrés actuellement dans le développement de code de simulation. Le chapitre 4 contient la description de l'approche *MDE4HPC* ainsi que les principes de l'ingénierie dirigée par les modèles qu'elle propose d'appliquer au développement de code de simulation. Le chapitre 5 porte sur le langage de modélisation *HPCML*, pilier de l'approche *MDE4HPC*, que nous avons défini afin que les numériciens puissent spécifier leurs applications de simulation numérique.

Dans la troisième partie (outillage et validation), nous évaluons les propositions faites dans la partie précédente. Dans le chapitre 6, nous détaillons le fonctionnement de l'outil *ArchiMDE* qui est une implémentation de l'approche *MDE4HPC*. Le chapitre 7 illustre l'approche *MDE4HPC* à l'aide d'un cas d'étude basé sur un code d'électromagnétisme. Dans le chapitre 8, nous évaluons l'approche *MDE4HPC* en nous basant en partie sur les techniques déjà validées dans la littérature.

La dernière partie (perspectives et conclusion) commence avec le chapitre 9 qui donne un aperçu des pistes de recherches découlant de l'avancement actuel de la définition de l'approche *MDE4HPC*. Finalement, le chapitre 10 clôt ce manuscrit sur un bilan de nos travaux.

Première partie

Contexte

Chapitre

2

Simulation numérique et calcul haute-performance

God is REAL
(unless otherwise declared in an explicit type statement
or in an implicit declaration).

B. Graham.

Sommaire

Lors de la conception d'une théorie ou d'un produit, il est parfois trop couteux, long, dangereux ou tout simplement impossible de réaliser toutes les expérimentations ou prototypes qui permettraient de valider les choix effectués. La simulation numérique en se basant sur des modèles physiques et mathématiques offre une alternative séduisante pour la validation de ces choix de conception. C'est pour cette raison que l'utilisation de la simulation numérique ne cesse de prendre de l'ampleur dans le monde. Certains États la considèrent comme un élément clé de la course à la compétitivité que leurs entreprises doivent mener sur la scène internationale [6, 205].

La prochaine étape majeure pour la communauté du calcul haute performance est le franchissement de l'*exascale*, soit la mise en production d'un supercalculateur capable d'effectuer 10^{18} opérations à virgule-flottante par seconde. Par rapport au *K Computer*, le supercalculateur le plus puissant en 2012 selon le classement des 500 plus gros calculateurs [154], il nous reste à améliorer les performances par un facteur 100 pour franchir ce seuil. Toutes les grandes puissances mettent en place des programmes de recherche visant à développer les composants matériels et logiciels nécessaires au franchissement de cette étape qui, conformément aux prévisions actuelles, devrait se produire aux alentours de 2020 [72].

2.1 Univers de la simulation numérique

2.1.1 De notre perception de la réalité à la machine

Le développement d'un logiciel de simulation numérique comportent plusieurs étapes qui sont présentées dans la figure 2.1. La première étape consiste à choisir un modèle physique permettant de représenter notre perception d'un objet ou d'un phénomène réels.

Pour une majorité de ces modèles, une résolution analytique n'est pas possible. Dans ce cas, on a recours à une technique de résolution approchée du système d'équations physiques. Afin de pouvoir exécuter le modèle mathématique de résolution par un ordinateur, il est nécessaire de créer une implémentation logicielle via un langage de programmation adapté à la machine cible. Et c'est finalement à l'issue de cette ultime étape que nous pouvons exécuter nos scénarios de simulation sur les calculateurs compatibles. Malheureusement, et nous y reviendrons en détails dans la section 2.1.3, il est existe une adhérence plus ou moins forte entre le logiciel de simulation et le calculateur cible.

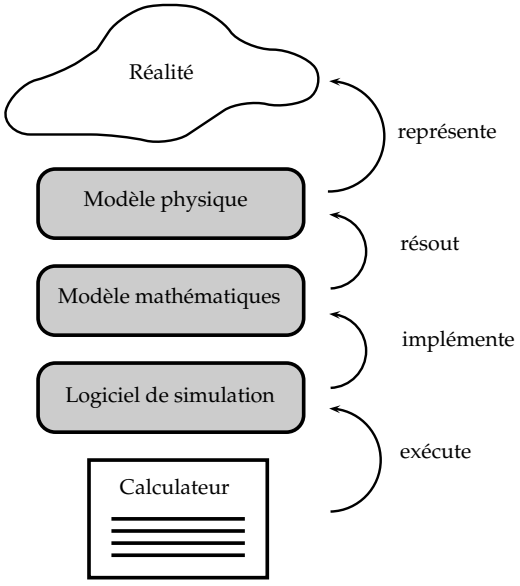


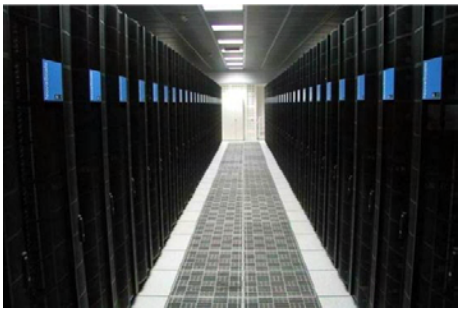
FIGURE 2.1 – Processus de développement d'un logiciel de simulation numérique

2.1.2 Programme Simulation

En 1996 la France a définitivement arrêté les essais nucléaires. Afin de garantir la fiabilité et la sûreté des têtes nucléaires sur le long terme, le programme *Simulation* a été mis en place [1]. Ce dernier vise à reproduire par le calcul les différentes phases de fonctionnement d'une arme nucléaire. Il se décompose en trois volets que sont : la physique des armes, la validation expérimentale et la simulation numérique qui nous intéresse plus particulièrement dans notre cas.

De la complexité et de la précision des modèles développés par les scientifiques du *CEA/DAM* découlent des besoins colossaux en puissance de calcul. Le programme Tera [99] a été mis en place afin de fournir les moyens de calcul nécessaires au volet simulation numérique du programme *Simulation*. Dans le cadre de ce dernier, le principal supercalculateur du *CEA/DAM* est remplacé tous les quatre ans avec un objectif d'accroissement des performances par un facteur supérieur à dix. La figure 2.2 présente le précédent modèle, Tera-10 ainsi que son successeur Tera-100 en production depuis juin 2012. Il est intéressant de noter deux faits parmi les chiffres présentés dans cette figure :

– *Performance* : en 1965 [158] Gordon Moore fit l'une des prédictions les plus visionnaires de toute l'histoire de l'informatique. Il modifia son propos en 1975 [159] et énonça que le nombre de transistors des microprocesseurs sur une puce de silicium doublerait tous les deux ans. Son propos originel a été altéré à maintes reprises et l'on considère de nos jours que la loi de Moore décrit le doublement de la performance



Tera-10 (52.8 TFlops)

Tera-100 (1.05 PFlops)

rang au *TOP500*	date	rang au *TOP500*
5e	11/2006	-
12e	06/2007	-
33e	06/2008	-
71e	06/2009	-
100e	11/2010	6e
265e	06/2011	9e

FIGURE 2.2 – Deux derniers supercalculateurs du programme Tera et leur classement respectif au *TOP500* [154]

tous les 18 mois.
À présent si l'on regarde le dernier changement de machines du programme Tera, on constate que les performances entre les deux versions ont été améliorées par un facteur 20 en 4 ans alors que sur la même période la loi de Moore prédit un facteur d'augmentation de 4. Avec un tel rythme, des ruptures technologiques matérielles apparaissent.

– *Obsolescence* : prenons l'exemple de la machine Tera-10, lors de son premier classement au *TOP500* [154]. Elle figurait parmi les cinq supercalculateurs les plus puissants au monde. Seulement six mois plus tard, elle se trouvait déjà à la 12e place du classement. Au bout de quatre ans et demi, elle était reléguée dans la deuxième partie du classement.
Face à cette obsolescence fulgurante, on comprend qu'un renouvellement fréquent est inéluctable, ne serait-ce que pour des raisons énergétiques, puisque l'évolution de la performance par watt des calculateurs suit une tendance proche de la loi de Moore [158] (qui s'applique seulement à l'évolution de la performance). En effet, on constate qu'un des aspects de ces machines qui revêt de plus en plus d'importance est la consommation électrique. À titre de comparaison, la puissance électrique nécessaire au *K Computer* évoqué précédemment est de 12,7 Mw soit plus que la production électrique d'un petit pays comme le Togo. Signe de la prise de conscience de l'importance de la consommation électrique, le *GREEN500* [86], un nouveau classement des supercalculateurs en fonction de leur performance par watt a été créé.

2.1.3 Limites du modèle actuel

Dans cette quête effrénée vers toujours plus de puissance de calcul, une question subsiste : serons-nous capable d'exploiter nos logiciels de simulation efficacement sur ces futures machines ?

Au cours de la dernière décennie, les fabricants de microprocesseurs se sont heurtés aux

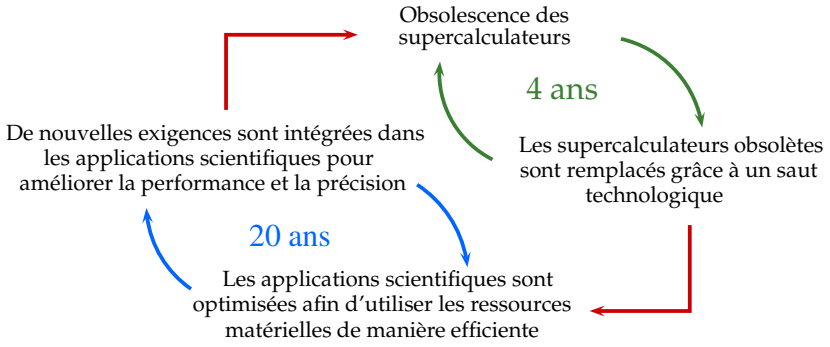


FIGURE 2.3 – Asynchronisme du cycle de vie des applications vis à vis de celui des supercalculateurs

limites physiques des technologies de gravure actuelles [185] avec pour conséquences de sérieux problèmes de dissipation de chaleur et de synchronisation de données. Ces limitations les ont contraint à revoir en profondeur la conception de l'architecture des processeurs. C'est pour cette raison que l'on a assisté à l'émergence d'architectures parallèles au sein même des microprocesseurs grand public : les processeurs multicœurs. L'hégémonie de ces nouvelles architectures sonne le glas du modèle de programmation séquentiel en tant que tel puisqu'il ne permet pas de tirer pleinement parti de la puissance de ces nouveaux processeurs. Ce changement de paradigme est en train de modifier profondément les habitudes de développement des logiciels courants.

A contrario, la communauté du calcul haute performance est coutumière de la programmation d'architectures parallèles puisque elles sont présentes dans les calculateurs depuis la fin des années 1980. Malheureusement les pratiques actuelles de programmation parallèle dans ce milieu se situent à un bas niveau d'abstraction. Bien qu'un bon niveau de performance puisse être atteint avec la plupart de ces approches, un certain nombre d'inconvénients apparaissent en termes de dépendances vis-à-vis de l'architecture, de mélange des préoccupations et de complexité de programmation :

Problème N° 1 **Dépendance vis-à-vis de l'architecture**

Dans le cas du *CEA/DAM*, l'expérience nous montre qu'en moyenne les modèles de simulation numérique associés à nos problématiques ont une espérance de vie de l'ordre de vingt à trente ans et doivent donc être maintenus sur cette période. De façon identique aux autres types de logiciel, toutes les contraintes liées à la maintenance logicielle sur de longues périodes doivent être prises en compte. On peut citer, par exemple, le renouvellement des équipes de développeurs ou l'obsolescence des technologies logicielles. Il y a cependant une contrainte propre à notre domaine qui est la nécessité de réaliser des *portages lourds avec une fréquence élevée*. En effet, dans la présentation du programme *Tera* (Partie 2.1.2) nous avons évoqué un changement de machines tous les quatre ans avec parfois des différences prononcées d'architecture. La figure 2.3 [143] résume clairement la situation, la durée de vie des ressources matérielles est cinq à sept fois plus courte que la durée de vie des applications propres à notre domaine.

Malheureusement les applications de simulation numérique sont mal armées face à ces fréquents changements. Comme nous allons le voir dans la partie 2.2, les architectures des supercalculateurs sont très spécifiques. L'utilisation de modèles de programmation très proches du niveau de la machine pour exploiter finement et avec efficacité ces spécificités induit une dépendance forte entre le logiciel et sa plateforme

d'exécution. C'est pour cette raison que la complexité de portage d'un logiciel de simulation entre deux supercalculateurs peut être élevé afin de conserver des propriétés de performance satisfaisantes.

On peut observer dans la figure 2.2 un effet collatéral provoqué par ces portages complexes : la machine Tera-10 est restée en production simultanément à Tera-100 pendant un an. La performance par watt et la puissance bien supérieure de Tera-100 suggèrent logiquement une mise hors tension rapide de Tera-10, or ce n'est pas le cas. Ce délai se justifie par la nécessité d'effectuer le portage entre machines et surtout de s'assurer de la validité du portage puiqu'il serait désastreux qu'un changement de machine provoque une dégradation à l'égard des fonctionnalités.

Problème N° 2 **Mélange des préoccupations/manque de formalisme**

Les différentes étapes du processus de développement d'un logiciel de simulation numérique ont été présentées dans la figure 2.1. L'implémentation logicielle du modèle mathématique de résolution (schéma numérique) contient à la fois des informations relatives aux modèles physiques et mathématiques ; et des informations relatives à la gestion de la plateforme d'exécution. Ces instructions concernant la partie purement informatique (gestion des données et du parallélisme) sont entrelacées avec des instructions concernant la résolution des modèles physiques. Cet enchevêtrement amène son lot de problèmes :

- la capitalisation sur les connaissances qui vise à sauvegarder le savoir-faire acquis par un individu lors de sa carrière dans une entreprise est impactée par le mélange des préoccupations et le manque de formalisme des techniques de développement actuelles qui ne fournissent pas une vision facilement compréhensible de la conception des différents artéfacts logiciels. Le *turn-over* des équipes de développement est problématique puisque une partie de la connaissance part avec la personne.
- les phases de portage présentées précédemment sont profondément affectées puisque les instructions concernant différents aspects sont complètement mélangées. Sachant que lors d'un changement de machine les modèles physiques et mathématiques n'ont pas de raison d'être modifiés, il est difficile d'identifier et de mettre à jour les parties informatiques sans altérer ces derniers.
- pour les mêmes raisons que la maintenance adaptative évoquée à l'instant, la maintenance corrective et la maintenance évolutive se voient compliquées.
- l'imbrication des différents aspects (mathématiques, parallélisme) provoque un manque de modularité qui freine la réutilisation entre projets.

Problème N° 3 **Complexité de programmation** Lors de l'établissement d'un profil type des développeurs d'applications de simulation une tendance émerge : la plupart d'entre eux ne possèdent pas un bagage informatique très important, notamment en génie logiciel puisqu'ils ont accompli des études longues en mathématiques appliqués ou dans le domaine d'application de la simulation. Ils sont par contre souvent familiers avec les langages dédiés très proche du formalisme mathématique tel que le langage MATLAB [4].

D'un autre coté le développement d'une simulation parallèle requiert beaucoup plus de connaissances informatiques qu'un développement standard de logiciel. Bien que nous aborderons le sujet plus en détails, il est judicieux de mentionner que l'on assiste aujourd'hui à une diversification des langages et bibliothèques de programmation parallèle en corrélation avec la diversification des architectures parallèles. Cette profusion de nouveautés restreint l'utilisation de ces nouvelles architectures à quelques développeurs ou scientifiques prêts à investir du temps dans l'apprentissage de leurs spécificités matérielles et logicielles.

Au final, on demande aux développeurs de simulation numérique l'assimilation de beaucoup de concepts qui les empêchent de se concentrer sur leur cœur de métier sur lequel ils pourraient apporter le maximum de valeur ajoutée.

2.2 Architectures des supercalculateurs

La compréhension des contraintes inhérentes aux applications de simulation numérique nécessite une connaissance des architectures matérielles et de leurs évolutions. Il n'est en effet pas possible de décorréler l'analyse des systèmes logiciels de celle des supports d'exécution. Cette partie s'attache donc tout d'abord à proposer une classification des différentes architectures matérielles tout en présentant les concepts clés du domaine. Après un rapide historique des types d'évolution auxquels nous avons assisté depuis les années 1960 jusqu'aux années 2000, nous terminerons par un tour d'horizon des évolutions possibles pour le futur.

2.2.1 Classification des architectures

Il existe plusieurs façons de catégoriser les différentes architectures matérielles. L'approche la plus courante se base sur la taxonomie définie par Flynn en 1972 [88]. Nous commencerons donc par présenter cette dernière. Cependant son manque de précision dans notre cadre nous amènera à introduire des concepts additionnels afin d'étendre sa portée.

Taxonomie de Flynn

La taxonomie de Flynn classifie les architectures en fonction de leur nombre de flots d'instructions et de flots de données. Sachant que cette taxonomie ne prend en compte que la différence entre flot unique et flots multiples, on obtient quatre combinaisons possibles :

– **Single Instruction, Single Data (*SISD*)** : cette architecture, présentée dans la figure 2.4, est la plus simple puisque la notion de parallélisme n'intervient au niveau ni des données, ni des instructions. Il n'y a qu'une seule unité d'exécution pour produire les résultats. Cette catégorie correspond à l'architecture dite de Von Neumann et définie par ce dernier dans son rapport [210] paru en 1945 sur la conception logique de l'ordinateur *EDVAC* (*Electronic Discrete Variable Automatic Computer*).

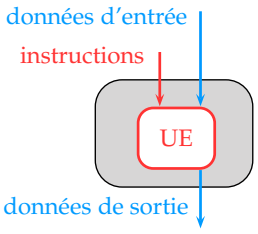


FIGURE 2.4 – Architecture *SISD*

– **Single Instruction, Multiple Data (*SIMD*)** : cette catégorie regroupe les architectures qui permettent d'appliquer un unique traitement (instruction) à un ensemble de données comme décrit dans la figure 2.5. Ces architectures sont donc particulièrement adaptées au calcul vectoriel. L'architecture actuelle des processeurs graphiques aussi appelés *graphics processing unit* (*GPU*) est partiellement basée sur cette approche.

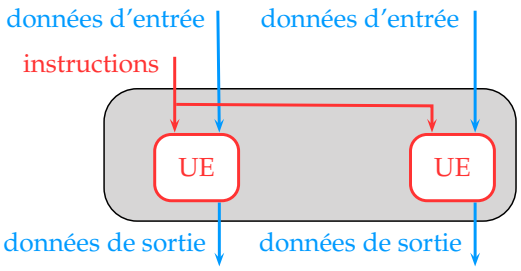


FIGURE 2.5 – Architecture *SIMD*

- **Multiple Instruction, Single Data (*MISD*)** : ce type d'architecture est très peu courant. En effet on cherche à appliquer des traitements multiples sur une donnée unique, les applications sont de ce fait limitées.
- **Multiple Instruction, Multiple Data (*MIMD*)** : on termine par la catégorie rencontrée le plus fréquemment. Effectivement la plupart des systèmes parallèles actuels peuvent être placés dans cette catégorie. Comme le met en avant la figure 2.6, ce type d'architecture comprend plusieurs unités d'exécution qui possèdent chacune leur propre flot de données. Cette définition permet beaucoup de marge de manœuvre et regroupe par conséquent des systèmes très variés. C'est pour cette raison que la prochaine partie s'attachera à caractériser différents types de sous-systèmes de cette catégorie.

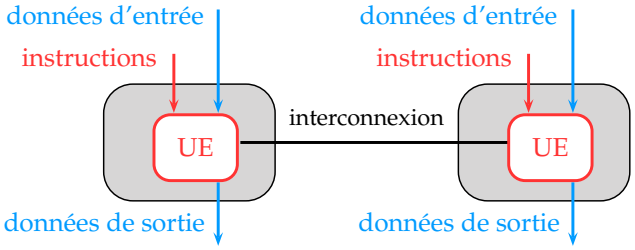


FIGURE 2.6 – Architecture *MIMD*

Classification étendue

Nous venons de voir que la taxonomie de Flynn n'est pas assez précise pour différencier les grandes classes de calculateurs. La figure 2.7 propose une vue d'ensemble d'une classification étendue, en particulier de la catégorie *MIMD* qui nous concerne plus particulièrement. Ainsi la catégorie *MIMD* peut se décomposer en deux grandes familles : les architectures où la mémoire est partagée et celle où la mémoire est distribuée.

Premièrement, dans le cas de l'architecture à mémoire partagée, décrite dans la figure 2.8, les différentes unités d'exécution accèdent à une mémoire commune et partagent donc le même adressage mémoire. De prime abord cette approche apparait intuitive mais elle est confrontée aux accès concurrents aux données (en lecture ou écriture) qui vont nécessiter l'introduction de nouveaux concepts tel que le sémaphore défini par Dijkstra [70]. Au sein des architectures à mémoire partagée, on trouve plusieurs sous-catégories mais nous n'en présenterons que deux : les *SMP* et les *NUMA*. Tout d'abord les machines *SMP* (*Symmetric MultiProcessing*) possèdent des temps d'accès à la mémoire identiques pour toutes les unités d'exécution. Malheureusement ce schéma simple du point de vue de l'utilisation

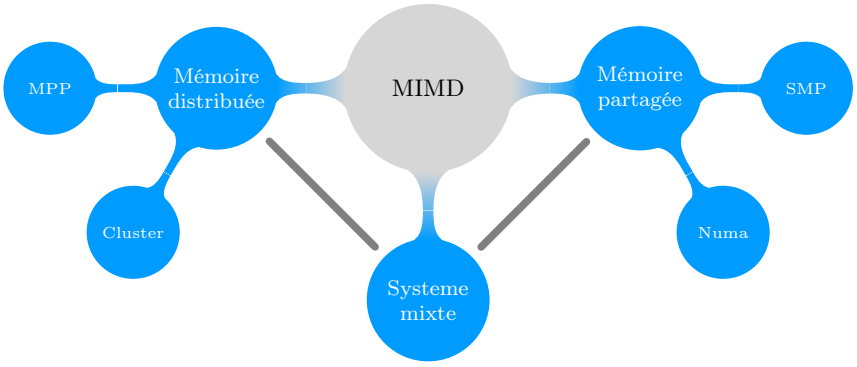


FIGURE 2.7 – Classification étendue des architectures matérielles

provoque un goulot d'étranglement sur l'accès à la mémoire lorsque l'on augmente le nombre d'unité d'exécution. Ensuite on trouve les machines où les accès mémoires ne sont pas uniformes, dites *NUMA* (*NonUniform Memory Access*), car elles possèdent une hiérarchie dans l'organisation de la mémoire. Par conséquent les temps d'accès à la mémoire dépendent de la proximité entre l'unité d'exécution et la mémoire à laquelle cette dernière souhaite accéder. Afin de pallier ce problème, l'architecture *cache coherent NUMA* (*ccNUMA*) propose d'améliorer l'architecture *NUMA* en ajoutant un cache à chaque unité exécution. Bien que les défauts de cache soient à prendre en compte, l'architecture *ccNUMA* se rapproche de l'architecture *SMP* d'un point de vue utilisation.

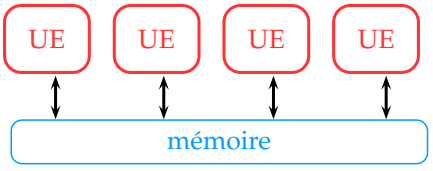


FIGURE 2.8 – Architecture à mémoire partagée

Deuxièmement, dans le cas de l'architecture à mémoire distribuée, décrite dans la figure 2.9, les différentes unités d'exécution possèdent leur propre mémoire et doivent communiquer par passage de messages avec les autres unités d'exécution pour accéder aux données distantes. L'architecture présentée ici repose donc sur l'assemblage d'éléments de base. Selon le type de ces éléments et de leur interconnexion, on différencie principalement deux sous-classes que sont les *massively parallel processors* (*MPP*) et les *clusters*. Ces deux types d'architectures sont les plus présents à l'heure actuelle parmi les grands supercalculateurs. Si l'on fait le parallèle avec l'industrie du textile, les *MPP* représentent la catégorie sur-mesure alors que les *clusters* font partie de la catégorie prêt-à-porter. En effet les machines *MPP* bénéficient d'une conception matérielle spécifique jusque dans l'architecture de ses briques élémentaires pour pouvoir fortement coupler les unités d'exécutions et leurs réseaux d'interconnexion. Cette conception sur-mesure permet certes d'atteindre d'excellentes performances mais au prix de couts de développement élevés. De leur côté, les *clusters* sont constitués d'un assemblage d'éléments plus standards et permettent donc de réduire le cout de leur développement.

Finalement il reste la classe des systèmes mixtes aussi appelés hybrides, qui mélangent plusieurs types d'architectures. Le cas le plus courant est le *cluster* de *ccNUMA*. On retrouve une architecture de type mémoire partagée au sein des nœuds de calcul et plus globalement

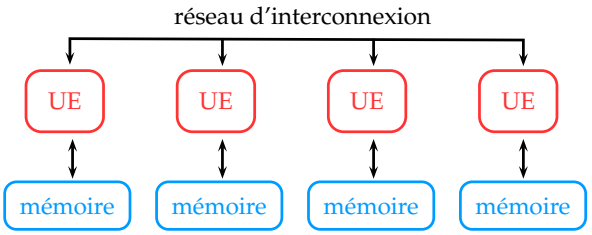


FIGURE 2.9 – Architecture à mémoire distribuée

entre les nœuds une architecture de type mémoire distribuée.

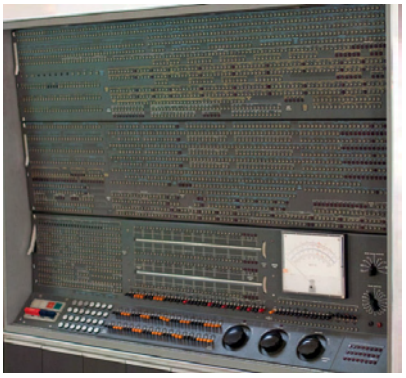
2.2.2 Évolution des architectures

À présent, nous allons retracer rapidement l'évolution des supercalculateurs et de leurs architectures matérielles. Bien que le passé ne préjuge pas de l'avenir, l'objectif de cette partie est de montrer que des changements sont apparus par le passé et que face aux challenges actuels la probabilité pour que l'avenir nous réserve des changements bien plus radicaux est très grande. Nous terminerons cette section par la présentation des architectures qui pourraient devenir courantes dans un futur plus ou moins proche.

Origine

L'automatisation du calcul scientifique fut la motivation première des pionniers de l'informatique. L'ouverture de l'informatique à d'autres domaines d'applications ne fut en effet que bien plus tardive.

En 1834 le mathématicien anglais Charles Babbage débute la conception d'une calculatrice programmable : l'*Analytical Engine*. Cette dernière, entièrement mécanique, ne sera malheureusement jamais terminée de son vivant. Il faudra attendre 1942 pour voir apparaitre le premier calculateur électronique : l'*ABC* (*Atanasoff Berry Computer*). Cette machine, capable de réaliser 30 additions en parallèle, n'était par contre pas programmable.



(a) *IBM 7030*

(b) *CDC 6600*

FIGURE 2.10 – Premiers supercalculateurs

Le premier ordinateur commercial scientifique avec virgule flottante, l'*IBM 704*, apparait aux États-Unis en 1955. C'est pour cet ordinateur que les langages de programmation *Fortran* et *LISP* ont initialement été développés. Cependant, on considère généralement

que l'ère des supercalculateurs a débuté dans le courant des années soixante avec la mise sur le marché de l'*IBM 7030* et du célèbre *CDC 6600* dont l'architecte était Seymour Cray [198] (cf. figure 2.10). Le *CDC 6600* restera le supercalculateur le plus puissant au monde pendant cinq ans. Son successeur, le *CDC 7600*, qui était dix fois plus puissant connaitra le même sort et restera numéro un pendant cinq ans.

Révolution vectorielle : l'ère Cray

Suite à des divergences d'opinions avec sa hiérarchie, Cray quitte *CDC* en 1972 pour fonder sa propre compagnie : *Cray Research*. Quatre ans plus tard, il met sur le marché le légendaire *Cray-1*, devenu l'un des plus grands succès de l'histoire des supercalculateurs [121] (cf. figure 2.11).

FIGURE 2.11 – Seymour Cray au coté d'un *Cray-1*

C'est le début d'une nouvelle ère, celle des architectures vectorielles. Ce type d'architecture propose des jeux d'instructions permettant de manipuler directement des vecteurs. La figure 2.12 compare les opérations nécessaires pour réaliser une addition de deux vecteurs A et B de taille 10 sur un processeur scalaire et sur un processeur vectoriel. L'implémentation des architectures vectorielles repose sur le principe du *pipelining*. À l'image du travail à la chaine imaginé par Henry Ford, le travail est découpé en étapes élémentaires qui peuvent être réalisées en parallèle.

pour *i allant de 1 à 10* **faire**
- Lire et décoder la prochaine instruction;
- Lire A(i);
- Lire B(i);
- Ajouter A(i) et B(i);
- Stocker le résultat dans C(i);

(a) Processeur scalaire

Lire et décoder la prochaine instruction;
Lire A;
Lire B;
Ajouter A et B;
Stocker le résultat dans C;

(b) Processeur vectoriel

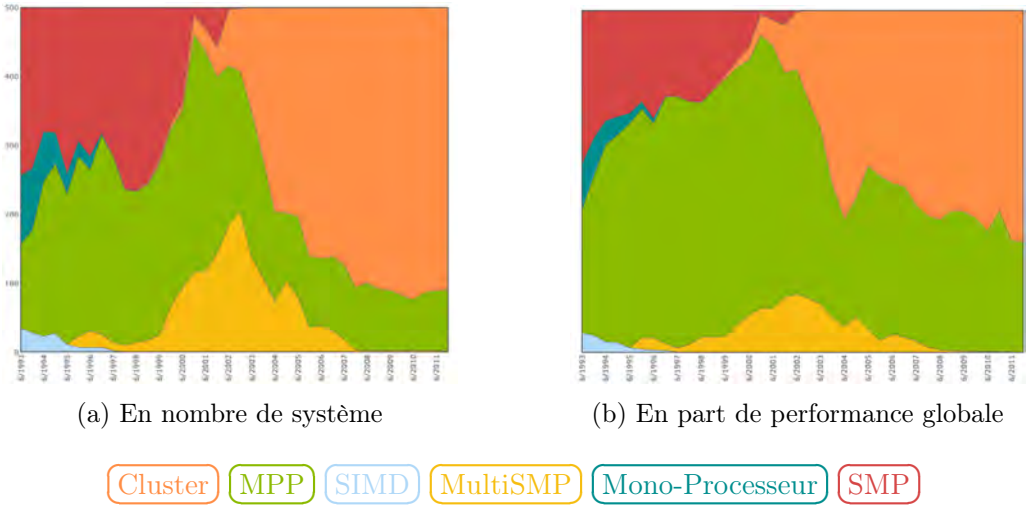
FIGURE 2.12 – Comparaison des opérations nécessaires pour réaliser une addition de deux vecteurs A et B de taille 10 sur un processeur scalaire et sur un processeur vectoriel.

En partie grâce aux calculateurs produits par *Cray Research*, les architectures vectorielles seront très populaires jusqu'à la fin des années 1980. Les machines de cette décennie

étaient principalement mono-processeurs, cependant des machines commerciales embarquant quelques processeurs (architecture *SMP* à base de processeurs vectoriels) firent leur apparition.

Ère des microprocesseurs et du massivement parallèle

À partir des années 1990 les limites du « modèle Cray » se font sentir et une nouvelle approche prônant l'ajout de parallélisme par interconnexion d'un grand nombre de processeurs émerge. Or les microprocesseurs grand public de l'époque offraient un rapport performance sur prix imbattable, il ne restait plus qu'à trouver un moyen efficace pour les faire communiquer. C'était le début d'une nouvelle révolution dans la conception des architectures des supercalculateurs : le massivement parallèle. Les deux sous-classes du *MIMD* gagnèrent en popularité, d'un coté on trouvait des machines à mémoire distribuée avec un grand nombre de processeurs et de l'autre des machines à mémoire partagée contenant un nombre plus restreint de processeurs.



(a) En nombre de système (b) En part de performance globale

FIGURE 2.13 – Évolution des types d'architecture du classement TOP500 depuis sa création [154]

La figure 2.13 montre l'évolution des parts des architectures du *TOP500* [154] depuis sa création en 1993. On y distingue la disparition des architectures purement *SMP* qui sont devenues trop couteuses et complexes lors de l'augmentation de leur taille. Dans le dernier classement de novembre 2011, on note clairement le succès rencontré par les *clusters*. Bien qu'en diminution, les *MPP* ne sont toutefois pas en reste puisqu'il représentaient 32% de la puissance pour seulement 17% du nombre des machines.

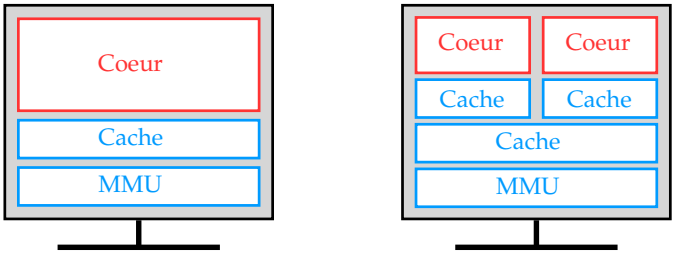
Avènement des processeurs multicœurs

Vers le milieu des années 2000, l'augmentation de la fréquence des processeurs n'était plus possible en raison des limites physiques évoquées en section 2.1.3. Les constructeurs ont toutefois profité de la réduction de la finesse de gravure pour ajouter des composants aux microprocesseurs : registres, *pipelines*, etc.

Les techniques dites de *simultaneous multithreading (*SMT*)* [203] tel que l'*hyper threading* d'*INTEL* [137] sont un exemple d'amélioration mise en œuvre par les constructeurs grâce au gain de place. L'architecture de ces processeurs *multithreads* vise à limiter le nombre de cycles d'horloge inutilisés pour cause d'attente. Ces attentes peuvent être dues

à des chargements de données en mémoire. Le fonctionnement de ces architectures repose sur le partage des *pipelines* entre plusieurs flots d'exécution : les *threads*. Cette approche, dans le meilleur des cas, permet de se rapprocher des performances théoriques maximales du processeur mais le cout de l'ordonnancement des différents *threads* peut devenir très lourd en termes de ressources de calcul. Dans le cas du calcul scientifique, les retours sur les gains de performance obtenus grâce à l'*hyper threading* sont mitigés. Alors que dans certains cas on observe des gains de performance notables [182, 21], dans d'autres cas on assiste à une détérioration de la performance [21]. Il existe plusieurs explications à ce phénomène : les *threads* d'un même pipeline utilisent les même types d'instructions, la mémoire cache est monopolisée par un seul *thread*.

Dans les années 1990, le modèle mono-processeur était devenu trop limité et du parallélisme massif a été introduit dans les calculateurs en multipliant le nombre de processeurs. Un parallèle avec les changements actuels peut être dressé. En effet, face aux limites des microprocesseurs séquentiels, la miniaturisation des composants est utilisée pour introduire du parallélisme au sein même de ces derniers en y incorporant plusieurs unités d'exécution appelées « cœur » [94]. On peut assimiler les architectures multicœurs au fait de rassembler plusieurs processeurs sur une même puce. La figure 2.14 présente les différences existantes entre les architectures des microprocesseurs mono et multicœurs. Il s'agit d'une simplification puisque pour un nombre plus élevé de cœurs il existe un hiérarchie complexe composée de plusieurs niveaux de caches. De cette organisation découle différents degrés d'affinité entre les cœurs.



(a) Microprocesseur séquentiel (b) Microprocesseur multicœur

FIGURE 2.14 – Comparaison des architectures des microprocesseurs mono et multicœurs.

Tendance à l'hétérogénéité

S'il est un sujet d'actualité dans la communauté du calcul haute performance, c'est bien celui de la percée des accélérateurs. Dans l'optique d'obtenir toujours plus de performance tout en prenant en compte la problématique énergétique, des accélérateurs sont accolés aux processeurs multicœurs habituellement présents au sein des nœuds des *clusters*. Nous n'aborderons ici que les accélérateurs les plus fréquemment rencontrés.

– **Graphics Processing Unit (*GPU*)** : à l'origine les processeurs graphiques ont été conçus pour épauler le processeur central dans le rendu des scènes 2D et 3D. Cependant leur prodigieuse évolution les a rendu capables de résoudre des systèmes algébriques complexes avec une grande efficacité. Ces processeurs possèdent en effet plusieurs centaines de processeurs de flux (*Streaming Processors*) permettant un parallélisme massif adapté au calcul scientifique. On parle alors de *General-Purpose computing on* GPU *(GPGPU)* [142]. Suite à l'engouement de la communauté du calcul scientifique vis-à-vis des *GPU*, les constructeurs améliorent les capacités pour le calcul de ces derniers en ajoutant des fonctionnalités : double précision, gestion de

l'*ECC (Error Correcting Code)* [214]. La figure 2.15 présente un exemple d'architecture de *GPU* : *Fermi* de *Nvidia*. On y distingue plusieurs *streaming multiprocessors (SM)* qui regroupent les *Cuda Cores* (carrés verts). Au total on compte pas moins de 512 *Cuda Cores*, chacun d'entre eux ayant un processeur de flux disposant de son propre *pipeline* d'exécution et de ses opérateurs.

FIGURE 2.15 – Architecture *Nvidia Fermi* du *GF100*

À l'image du processeur *Cell BE* que nous allons présenter par la suite, la conception de processeurs hétérogènes est très en vogue. Le projet *Fusion* [44] du fondeur *AMD* en est un parfait exemple puisque le processeur renommé pour l'occasion *Accelerated Processing Unit* (*APU*) embarque à la fois des cœurs généralistes et des cœurs « graphiques » de type *SIMD* (cf. figure 2.16).

FIGURE 2.16 – Organisation d'un *AMD Fusion APU*

– **Cell Broadband Engine (*Cell BE*)** : issu de la collaboration entre *IBM*, *Sony* et

Toshiba, ce processeur repose sur une architecture hétérogène [172]. Il possède en effet un cœur généraliste appelé *PowerPC Processing Element* (*PPE*) qui est accompagné de 8 cœurs spécialisés : les *Synergitic Processor Elements* (*SPE*). L'ensemble de ces éléments communique par l'intermédiaire d'un bus appelé *Element Interconnect Bus* (*EIB*). La figure 2.17 fournit un aperçu de l'architecture du *Cell BE* avec l'ensemble de ces éléments. Le *PPE* qui joue un rôle de factotum et de manager pour les *SPE* possède une conception se rapprochant d'un processeur *PowerPC* 64 bits simplifié. Par exemple, il n'est pas capable d'effectuer des exécutions de type *out of order* dont les composants occupent traditionnellement beaucoup de place sur une puce. Les *SPE*, quant à eux, sont basés sur le principe du *SIMD* et sont donc très performants pour le calcul vectoriel.

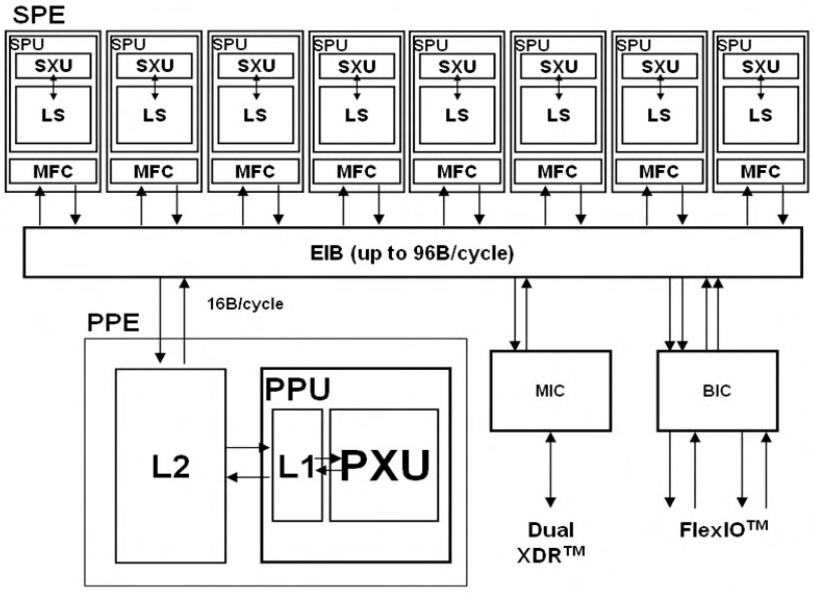


FIGURE 2.17 – Architecture du *Cell BE*

Le microprocesseur *Cell BE* est utilisé dans de nombreux domaines autres que le calcul haute-performance : jeux vidéos (*Playstation 3*), multimedia (TVHD), traitement du signal. L'*IBM Roadrunner* [20] est certainement le supercalculateur utilisant des *Cells* le plus célèbre puisqu'il a été le premier ordinateur à franchir le seuil du PetaFlops. Ce dernier possède une architecture hétérogène : il est composé de 6562 processeurs bicœurs *AMD Opteron* couplés à 12240 *IBM PowerXCell 8i*, une variante du *Cell BE* classique. Dans cette configuration, les *Cells* sont utilisés comme accélérateur par les *Opterons*.

Innovation de rupture

A moyens termes, nous ne sommes pas à l'abri de changements radicaux dans la conception des architectures des supercalculateurs : puce 3D, ordinateur quantique, électronique moléculaire. Que pourrions-nous récupérer des logiciels de simulation actuels ? Où trouver et comment extraire l'information nécessaire à un portage vers ces hypothétiques machines ?

2.2.3 Bilan

En prenant en compte la tendance actuelle qui nous amène vers plus d'hétérogénéité, nous avons dû faire face à cinq changements majeurs dans la façon de concevoir les architectures des supercalculateurs lors des cinquante dernières années. Or ces ruptures technologiques, qui arrivent donc en moyenne tous les dix ans, peuvent avoir des répercussions énormes sur les applications de simulation numérique. Bien qu'une partie de ces applications puisse toujours être exécutée sur les nouvelles machines, toutes ne sont pas capables d'exploiter la puissance de calcul offerte par ces dernières. Nous verrons dans le prochain chapitre dans quelle mesure les méthodes et outils disponibles actuellement sont aptes à gérer cette situation.

Chapitre

3

Développement d'applications de calcul scientifique

La connaissance s'élabore contre une connaissance antérieure.

Gaston Bachelard.

Sommaire

Le chapitre précédent nous a permis d'aborder à la fois la diversité et le caractère éphémère des architectures matérielles que l'on retrouve au sein des supercalculateurs. Le présent chapitre a pour objectif de présenter la manière dont les logiciels de simulation numérique sont développés afin de pouvoir exploiter ces différentes architectures matérielles. Cette étude vise à mettre en exergue les lacunes de chacune de ces solutions vis-à-vis des trois problèmes soulevés dans la section 2.1.3 : dépendance à l'architecture, mélange des préoccupations et manque de formalisme, complexité de programmation.

La section 3.1 présente tout d'abord un panel de langages et de bibliothèques conçus pour le développement de logiciels parallèles. Nous essayerons d'évaluer plusieurs propriétés de ces approches. La section 3.2, en se consacrant plutôt sur l'aspect processus, explore la relation qui lie calcul scientifique et génie logiciel.

3.1 Langages et outils

Bien qu'elle n'ait pas la prétention d'être exhaustive, cette section présente un large panel de langages et bibliothèques utilisés pour le développement d'applications de calcul scientifique parallèle. Nous verrons que l'expression du parallélisme revêt plusieurs formes. C'est pourquoi nous choisissons le terme « solution logicielle » comme dénominateur commun à toutes les formes possibles : langages, bibliothèques, directives de compilation, langage dédié embarqué, etc.

Cette section a pour objectif de relever les forces et faiblesses de chacune de ces solutions logicielles vis-à-vis de notre problématique. Afin de mettre en avant les catégories de problèmes les plus récurrents, nous avons défini cinq propriétés : la séparations des préoccupations, la validation, la portabilité, l'abstraction et l'accessibilité, et finalement l'aspect communauté et outils. La complétude de chaque propriété sera notée de 1 (faible) à 3 (forte) en fonction des critères définis en annexe A.

3.1.1 Fortran : l'origine

Le langage *Fortran*, abbréviation de *FORmula TRANslator* [56], possède des origines qui se confondent avec celles des supercalculateurs. C'est en effet pour faciliter la programmation de l'*IBM 704* (voir section 2.2.2) que J. Backus proposa un langage de plus haut niveau que les traditionnels langages assembleur utilisés à cette époque. Il décrivait alors *Fortran* comme étant un système de programmation automatique [16]. La définition de la programmation automatique a évolué au cours du temps mais D. Parnas avec un brin d'humour la synthétise ainsi :

> *« Automatic programming always has been a euphemism for programming with a higher-level language than was then available to the programmer »*
> D. Parnas [170]

Depuis plus d'un demi-siècle, le langage *Fortran* est la référence de la communauté scientifique pour le développement d'application de simulation numérique [84]. De son succès et de son ancienneté découlent une collection impressionnante de bibliothèques dédiées au calcul numérique. Dans le cas du *Fortran 90* qui reste la version la plus utilisée, nous avons affaire à un langage procédural qui offre pour les besoins du calcul numérique, des fonctions intrinsèques permettant de manipuler simplement des vecteurs et des matrices. Les premières versions ont su s'adapter d'elles mêmes aux deux premières périodes présentées dans la section 2.2.2. Dans le cas des deux dernières périodes (ère multiprocesseur et ère multicœur), il est nécessaire de faire appel à des bibliothèques externes dont certaines seront présentées par la suite.

Bibliothèque de fonctions métiers

Les logiciels de simulation numérique sont composés de certaines briques de base comme la résolution de systèmes linéaires, l'intégration de fonctions ou l'application de transformés de Fourier. Des bibliothèques regroupant des fonctions optimisées permettant d'accomplir ce type de tâche sont légion : *LAPACK* [12], *PLAPACK* [207], *FFTW* [92], *MKL* [3], *NAG* [171], *Hypre* [82], *PETSc* [17]. C'est d'ailleurs une des forces du langage *Fortran* qui en compte une grande quantité. À ce stade, il faut néanmoins noter que le couple *C/C++* possède aussi un grand nombre de ce type de bibliothèques et que le *C++* gagne en importance dans la communauté scientifique. Nous pourrons d'ailleurs constater ce

phénomène dans la suite de ce chapitre puisqu'un grand nombre de solutions logicielles sont compatibles avec ou basées sur le langage *C++*.

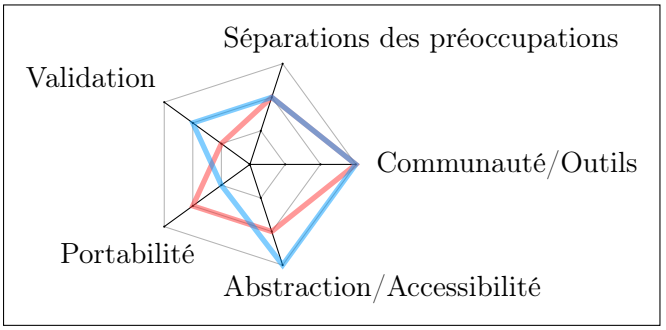
Une des solutions pour commencer la parallélisation d'un code de simulation est d'utiliser les versions parallèles de ce type de bibliothèque. Le parallélisme est alors exprimé au sein de ces bibliothèques grâce aux solutions logicielles que nous présentons dans la suite du chapitre. Pour ces raisons, ces bibliothèques ne sont donc compatibles qu'avec certaines classes d'architectures. Cette approche possède un certain nombre d'avantages et d'inconvénients :

+ les bibliothèques ont l'avantage d'abstraire les technologies sous-jacentes et enlève ainsi à l'utilisateur le besoin de se former sur de nouvelles solutions logicielles. Par conséquent, un développeur peut rapidement paralléliser son code sur une architecture dont il ne maitrise pas les subtilités.
+ les bibliothèques sont souvent développées par des experts des méthodes contenues dans ces bibliothèques. Ils ont donc la capacité de produire du code très optimisé ;
− chaque bibliothèque possède ses propres structures de données. Il faut donc soit ajouter une étape de conversion des données avant l'appel à la bibliothèque, soit utiliser la structure de données de la bibliothèque dans tout le programme. Même si la seconde solution semble préférable, deux problèmes sont présents. Dans le cas où plusieurs bibliothèques sont utilisées, on ne pourra passer outre les étapes de conversion des données. Deuxièmement, l'utilisation de la structure de données d'une bibliothèque posera des problèmes lorsque l'on souhaitera essayer une autre bibliothèque ou tout simplement porter l'application sur une autre architecture matérielle ;
− ce type d'approche ne permet qu'une parallélisation locale à une étape de la simulation.

En résumé l'utilisation de bibliothèques de fonctions métiers est utile pour paralléliser une partie spécifique d'un code de simulation mais ne fournit pas d'approche globale de parallélisation.

Évaluation de la catégorie

Légende :
- *Fortran*
- bibliothèques métiers



3.1.2 Passage de messages

Plusieurs bibliothèques de communications se fondent sur le paradigme de communication par passage de messages. Des tâches sont affectées aux différentes unités d'exécution et communiquent par des envois explicites de données (les messages). Cette approche est donc particulièrement adaptée aux architectures de type mémoire distribuée (voir section 2.2.1). On distingue deux types de communications :

- *communications point à point* : ce type de communication correspond à l'échange de messages d'un processus émetteur à un processus récepteur. Ces communications

sont définies de façon explicites en utilisant l'identifiant des processus ;
- *communications collectives* : ce type de communication permet l'envoi de messages à un groupe de processus. Il existe trois grandes catégories d'opérations collectives : synchronisation (ex : barrière), communication (ex : diffusion) et calcul (ex : réduction).

De plus ces communications peuvent être soit de type bloquantes (approche synchrone) soit de type non bloquantes (approche asynchrone). Ces bibliothèques reposent principalement sur le modèle d'exécution *single program, multiple data* (*SPMD*), c'est à dire que tous les processus exécutent le même code qui est paramétré en fonction de l'identifiant du processus. Ce modèle provoque un fort couplage de l'algorithmique métier avec la gestion du parallélisme. Nous allons maintenant présenter les deux principales solutions logicielles basées sur cette approche.

Parallel Virtual Machine (PVM)

Conçu initialement en 1989 par le *Oak Ridge National Laboratory*, *PVM* [197, 95] est une bibliothèque de communications par passage de messages destinée aux architectures à mémoire distribuée. Elle a pour objectif d'exploiter des ensembles de machines hétérogènes reliées par des réseaux d'interconnexions. *PVM* est utilisable par les langages *C*, *C++* et *Fortran*. Bien qu'elle connût le succès à ses débuts, la bibliothèque *PVM* s'est fait supplanter par le standard *MPI* dans le courant des années 1990 notamment pour des questions de performances.

Message Passing Interface (MPI)

MPI est certainement l'approche la plus utilisée à l'heure actuelle pour le développement d'applications de calcul scientifique parallèle. Contrairement à *PVM*, *MPI* [192] est issu d'un effort de standardisation regroupant des développeurs et des constructeurs. Sa première version fut publiée en juin 1994 [89]. Le listing 3.1 montre un exemple tiré du livre « *Using MPI* » [105]. Cet exemple, écrit en langage *C* en utilisant le standard *MPI*, calcule les décimales de π par la méthode des trapèzes. Ce dernier montre en quoi *MPI* est un exemple typique de solution logicielle possédant une vue locale tel que définie par Chamberlain (voir annexe A) : l'algorithme y est défini pour chaque processus et les échanges de données sont explicites. La maitrise fine des communications ajoute de la complexité ainsi qu'une grande verbosité si l'on établit une comparaison avec des solutions logicielles à vue globale [50].

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <math.h>
4  int main( int argc, char *argv[] ) {
5      int n, myid, numprocs, i;
6      double mypi, pi, h, sum, x;
7      // Initialisation de l'envirronement MPI
8      MPI_Init(&argc,&argv);
9      // Détermine le nombre total d'unités d'exécution
10     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
11     // Détermine le rang (numéro) de l'unité d'exécution courante
12     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
13     while (1) {
14         // L'unité d'exécution 0 se charge des entrées/sorties
15         if (myid == 0) {
16             printf("Enter the number of intervals: (0 quits) ");
17             scanf("%d",&n);
18         }
```

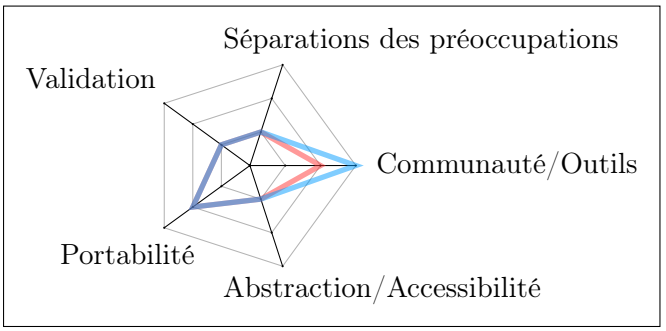
```
19          /* Transmet le nombre d'intervalles sélectionné par
20          l'utilisateur à toutes les unités d'exécution */
21          MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
22          if (n == 0)
23              break;
24          else {
25              h   = 1.0 / (double) n;
26              sum = 0.0;
27              for (i = myid + 1; i <= n; i += numprocs) {
28                  x = h * ((double)i - 0.5);
29                  sum += (4.0 / (1.0 + x*x));
30              }
31              mypi = h * sum;
32              /* Agrège les valeurs caculées sur chaque unité
33              d'exécution pour obtenir la valeur de PI */
34              MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
35                         MPI_COMM_WORLD);
36              if (myid == 0)
37                  printf("pi is approximately %.16f", pi);
38          }
39      }
40      MPI_Finalize();
41      return 0;
42  }
```

Listing 3.1 – Exemple de programme *MPI* écrit en *C* : calcul de π

Le standard *MPI* définit la syntaxe et la sémantique d'un ensemble de routines mais n'impose aucune caractéristique sur le matériel. Ce choix a permis l'émergence de plusieurs implémentations qui sont plus ou moins performantes selon les plateformes matérielles. Parmi les plus connues on retrouve *OpenMPI* [93] et *MPICH* [104]. Certains constructeurs livrent même des implémentations sur-mesure pour leurs machines. C'est le cas pour *Tera100*, puisque le constructeur *Bull* a livré avec la machine une implémentation optimisée de *MPI*, *Bull MPI* qui est basée sur l'implémentation *OpenMPI*. Les implémentations sont le plus souvent compatibles avec les langages *C*, *C++* et *Fortran* mais on trouve aussi des implémentations pour des langages tel que *Java* [41] qui sont traditionnellement moins présents dans le monde du calcul haute-performance. L'existence d'implémentations pour un grand nombre de langages et de plateformes est une des grandes forces du standard *MPI*.

Évaluation de la catégorie

Légende :
- *PVM*
- *MPI*



3.1.3 Mémoire partagée

Il existent plusieurs solutions logicielles dont la conception découle des spécificités des architectures à mémoire partagée. Nous n'aborderons ici que les principales : les *threads*, *OpenMP* et *TBB*.

Thread

Un *thread* est un processus léger. Il existe plusieurs bibliothèques permettant de gérer les différents aspects de leur cycle de vie : création, synchronisation et destruction. Historiquement les constructeurs fournissaient leur matériel avec des implémentations propriétaires qui permettaient de gérer ces processus légers. Ces implémentations, qui possédaient parfois des interfaces très différentes, rendaient l'écriture d'applications portables très ardue. Afin d'unifier ces différentes interfaces, le standard *IEEE POSIX 1003.1c-1995* fut proposé. Pour les implémentations qui respectent le standard on parle alors de *POSIX threads* ou *Pthreads* [45].

Traditionnellement on décompose les bibliothèques de *threads* en trois catégories. Tout d'abord, il y a les bibliothèques se trouvant dans l'espace utilisateur tel que *GNU Pth* (*Portable Threads*) [79]. Ces dernières étant au-dessus du système d'exploitation, elles ne peuvent pas exploiter les machines multiprocesseurs. Ensuite on trouve, les bibliothèques au niveau du noyau du système d'exploitation tel que *NPTL* [74]. Dans ce cas, c'est donc l'ordonnanceur de l'*OS* qui va gérer le cycle de vie des processus légers. Finalement, il existe des solutions mixtes telle que *Marcel* [62] où les deux types d'ordonnanceurs, niveau utilisateur et niveau noyau, cohabitent dans le but de permettre plus de flexibilité.

Bien qu'il n'y ait pas de communications à gérer, il persiste un problème de cohérence entre les données partagées entre les *threads* et celles se trouvant dans les registres de chaque *thread*.

Open Multi-Processing (OpenMP)

OpenMP [61, 53] est une interface de programmation qui est exploitée via l'utilisation de directives de compilation au sein du code source. Les spécifications d'*OpenMP* sont gérées par le consortium *OpenMP Architecture Review Board* qui est composé de grands noms parmi les sociétés de l'industrie logicielle et matérielle. Un compilateur compatible *OpenMP*, grâce aux informations fournies par les directives de compilation, produit des binaires *multithread* qui reposent le plus souvent sur la bibliothèque de *threads POSIX* fournie avec le système d'exploitation.

L'interface *OpenMP* est définie pour les langages *C*, *C++* et *Fortran*. Les directives de compilation sont présentes sous forme de *pragma* en *C* et *C++*, et sous forme de commentaires en *Fortran*. Cette approche permet de partir d'un code séquentiel existant et de rajouter par étapes successives des directives qui vont permettent de paralléliser l'application. Dans le cas où l'on utilise un compilateur n'implémentant pas *OpenMP*, les directives seront tout simplement ignorées et l'on obtiendra un binaire séquentiel. Le listing 3.2 montre un exemple de cette souplesse qui est permise dans le développement.

```
1 #define RADIUS 1000000000
2 using namespace std;
3
4 int main(int argc, char **argv){
5         double totalSum = 0;
6         double invRadius = 1.0/RADIUS;
7
```

```
8           #pragma omp parallel reduction( + : totalSum ){
9                   double partialSum = 0.0;
10                  double perc = 0.0;
11
12                  #pragma omp for
13                  for(int step = 0; step < RADIUS; step++){
14                          perc = (step + 0.5) *invRadius;
15                          partialSum += sqrt(1 - pow(perc,2))*invRadius;
16                  }
17                  totalSum += partialSum;
18          }
19
20          cout << "Pi: " << (totalSum*4) << endl;
21          return 0;
22  }
```

Listing 3.2 – Exemple de programme *OpenMP* écrit en *C++*: calcul de π

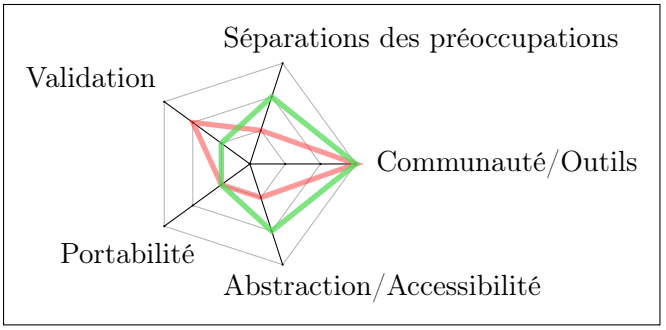
Intel Threading Building Blocks (TBB)

TBB [177] est une bibliothèque *C++* développée par la société *Intel* pour exploiter la puissance des processeurs multicœurs. *TBB* vise à élever le niveau d'abstraction par rapport aux *threads* en automatisant certains aspects liés à leur manipulation : création, synchronisation et destruction. À cet effet, *TBB* propose de décrire les applications sous forme de tâches et de les appeler via une combinaison de *templates* parallèles (*parallel_for*, *parallel_reduce*, *parallel_scan*, *parallel_sort*, *parallel_while*). C'est grâce notamment à l'utilisation de ces *templates* parallèles que *TBB* est capable d'inférer les dépendances entre les tâches.

Évaluation de la catégorie

Bien que les architectures à mémoire partagée enlèvent le besoin d'expliciter les communications, il n'en reste pas moins que la cohérence des données entre la mémoire centrale et la mémoire des *threads* est difficile à gérer dans certains cas.

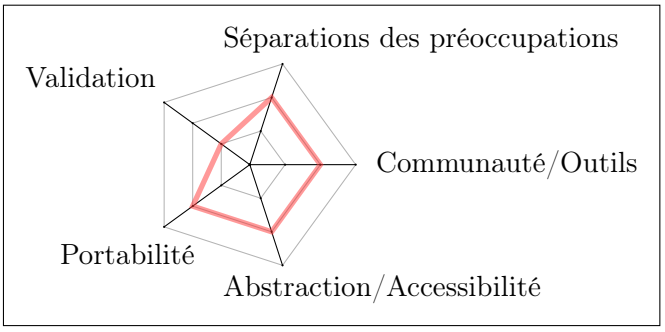
Légende :
- *Thread*
- *OpenMP*
- *TBB*



3.1.4 High Performance Fortran (HPF)

High Performance Fortran [135] est une extension du langage *Fortran 90* issue d'un effort de standardisation dans les années 1991 à 1993 par un groupe d'universitaires et d'industriels réunis au sein du *High Performance Fortran Forum*. L'idée centrale d'*HPF* est de pouvoir compléter un programme *Fortran* classique en ajoutant des directives afin de spécifier la distribution des données sur des mémoires disjointes [152]. C'est ensuite le compilateur qui a la charge de gérer les communications nécessaires. *HPF* introduit aussi

des constructions additionnelles tel que l'expression *FORALL*. Cette dernière permet de spécifier un calcul irrégulier, c'est-à-dire dont le contenu peut être exécuté dans n'importe quel ordre et donc simultanément pour exploiter le parallélisme. Bien que le langage *HPF* ne fut pas adopté massivement dans les vingt dernières années, plusieurs de ses concepts (spécification globale de la distribution des données, opérations primitives dédiées au parallélisme, techniques de compilation) eurent une influence notable sur de nombreux langages parallèles conçus après sa sortie [127].



3.1.5 Partitioned Global Address Space (PGAS)

Le modèle à adressage global partitionné (*PGAS*) repose sur le principe illustré dans la figure 3.1 : chaque processus possède un espace mémoire privé ainsi qu'un espace mémoire partagé via un adressage global accessible par tous les processus. Si l'on prend l'exemple présenté dans la figure 3.1, chaque processus possède une variable x dans sa mémoire privée. Le processus 1 possède une variable partagée y. Finalement un tableau A est réparti sur l'ensemble de l'adressage global. Bien que le modèle *PGAS* propose un adressage global, la localité des données (un processus a une plus grande affinité avec la mémoire partagée qu'il héberge) ainsi que leur partitionnement sont gérés de façon explicite. Les solutions logicielles basées sur le modèle *PGAS* adoptent une approche *SPMD* (voir section 3.1.2).

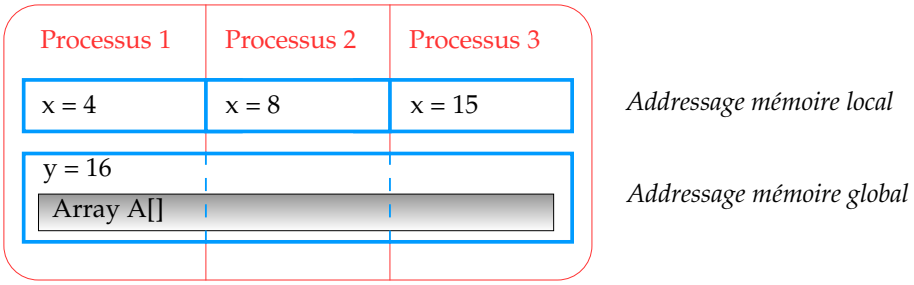


FIGURE 3.1 – Modèle *PGAS*

CoArray Fortran (CAF)

Coarray Fortran est une extension du langage *Fortran* basée sur le modèle *PGAS*. Chaque processus, appelé *image* dans *CAF*, possède des variables « traditionnelles » qui sont privées. Les variables qui possèdent une *codimension* sont accessibles depuis toutes les *images*. Cependant il existe dans le langage des constructions différentes pour accéder à une partie d'une variable partagée selon qu'elle soit hébergée ou non dans la zone mémoire partagée du processus qui en demande l'accès. Selon Reid [176], l'objectif de *CAF* était d'apporter au langage *Fortran* le minimum de modifications possibles pour qu'il devienne un

langage parallèle robuste et efficace. Cette extension a été intégrée dans la norme *Fortran 2008* (*ISO/IEC 1539-1 :2010*) [56].

Unified Parallel C (UPC)

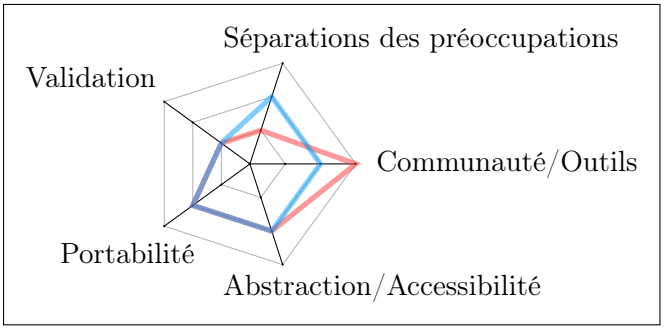
UPC [204] est une extension du langage *C* qui est basée sur le modèle *PGAS*. Comme pour *CAF* chaque processus partage une partie de l'espace mémoire. Le langage *UPC* demande au programmeur de contrôler la distribution des données entre les différents processus afin qu'il affecte à un processus donné, les données qu'il risque de manipuler le plus souvent. À la différence de *CAF*, *UPC* propose l'abstraction d'un adressage « plat » dans le lequel on peut accéder uniformément, modulo un surcout, à tous les éléments d'une structure de données partagées. La gestion des communications est donc déléguée au compilateur.

Dans [46], Cantonnet et coll. analysent la productivité du langage *UPC* à l'aide de métriques (nombre de lignes et de caractères dans le code source). Ils montrent que cette dernière est meilleure que celle du couple *Fortran*/*MPI*. Ils expliquent cette différence par le fait que les concepts du langage *UPC* sont plus proches de la façon de penser des développeurs que ne le sont les concepts proposés par le paradigme par passage de messages.

Évaluation de la catégorie

Le modèle *PGAS* est statique (un thread par partition mémoire), le modèle *asynchronous PGAS* (*APGAS*)tente de remédier au problème en proposant plus de dynamicité. D'ailleurs il y a une tentative pour appliquer le modèle *APGAS* au langage *UPC* [189]. Nous reviendrons plus en détails sur le modèle *APGAS* dans la section 3.1.6

Légende :
- *Coarray Fortran*
- *Unified Parallel C*



3.1.6 Initiative *HPCS* du DARPA

C'est en 2002 que l'agence américaine *DARPA* (*Defense Advanced Research Projects Agency*) lança un projet de grande envergure, le *High Productivity Computing Systems* (*HPCS*) [73]. L'objectif du projet est d'améliorer la productivité de la communauté du calcul haute-performance dans le but de servir les intérêts de la sureté nationale et des grands groupes industriels américains. Dans le cadre de ce projet, la productivité fut définie comme étant une combinaison de la performance, de la programmabilité, de la portabilité et finalement de la robustesse. Lors de la création du projet il était reconnu qu'un frein majeur à l'adoption à grande échelle de la simulation numérique était la complexité des développements logiciels visant à exploiter la puissance des machines de l'époque. Afin d'améliorer la productivité de ces systèmes de calcul, trois constructeurs de calculateurs (*Cray*, *IBM* et *Sun*) se sont penchés dans le cadre de ce projet autant sur les aspects matériels que logiciels en définissant de nouveaux langages parallèles. Le projet *Fortress* [11] de la société *Sun* n'ayant pas été retenu pour la dernière phase du projet *HPCS*, nous

nous concentrerons ici sur les projets de ses deux concurrents : *Chapel* de *Cray* et *X10* d'*IBM*.

L'ensemble de ces langages est basé sur le modèle *Asynchronous Partitioned Global Address Space* (*APGAS*) [184] qui est une extension du modèle *PGAS* présenté en section 3.1.5. Le modèle *APGAS* fournit un modèle d'exécution plus riche que le modèle *SPMD* que l'on retrouve généralement dans les langages de type *PGAS*. On retrouve la notion d'asynchronisme avec la possibilité d'exécuter plusieurs tâches simultanément sur une partition de mémoire partagée (dénommée *place* en *X10* et *locale* en *Chapel*) alors qu'il n'y a qu'un seul processus par partition mémoire dans le modèle *PGAS*. De plus, un processus peut invoquer l'exécution d'une tâche sur une partition de mémoire partagée autre que la sienne.

Chapel

Bien que sa syntaxe hérite de langage tel *C*, *Fortran*, *Java* ou *Ada* ; le langage *Chapel* [49] n'est une extension d'aucun de ces langages et possède ses propres constructions, syntaxe et sémantique. Les développeurs de chez *Cray* pensent que l'utilisation d'un langage séquentiel comme base pour la conception d'un langage parallèle mènerait à la confusion et encouragerait un style de programmation séquentiel [213]. Les concepts du langage *Chapel* sont inspirés du langage *HPF* (voir section 3.1.4) et du langage *ZPL* [51] sur lequel ont travaillé de nombreux développeurs de *Chapel*.

X10

Le nom du langage *X10* [54] provient du souhait de ses développeurs de mettre au point un langage parallèle dix fois plus productif (selon la définition de la productivité fournie en début de chapitre) que les langages et bibliothèques qui étaient les plus utilisés à l'époque. Contrairement au langage *Chapel* [49], le langage *X10* n'a pas été conçu en partant de zéro puisqu'il est une extension du langage *Java*. Le choix du langage *Java* par les développeurs d'*IBM* fut un choix pragmatique. En effet, c'est un langage orienté objet répandu qui possède tout un écosystème logiciel, bibliothèques et outils, dont certains sont développés par *IBM*.

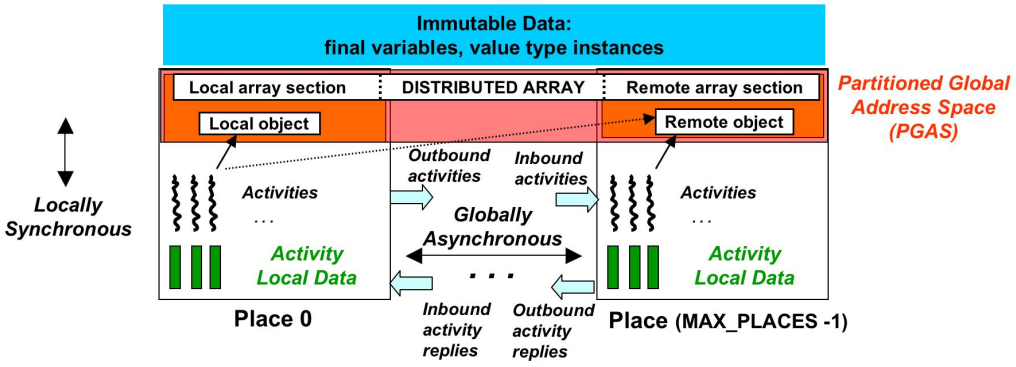
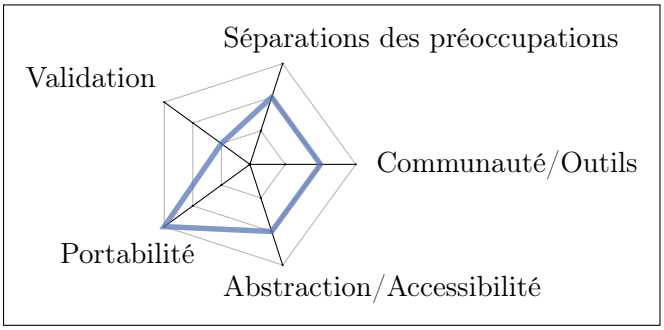


FIGURE 3.2 – Aperçu des concepts du langage *X10* [54]

En addition de la partie séquentielle traditionnelle du langage *Java*, *X10* propose de nouveaux types de données, accompagnés d'opérations de haut niveau pour les manipuler, et des concepts permettant d'exprimer le parallélisme : *places*, *activities*, *clocks*, type de données distribué (figure 3.2).

Évaluation de la catégorie

Légende :
- *Chapel*
- *X10*



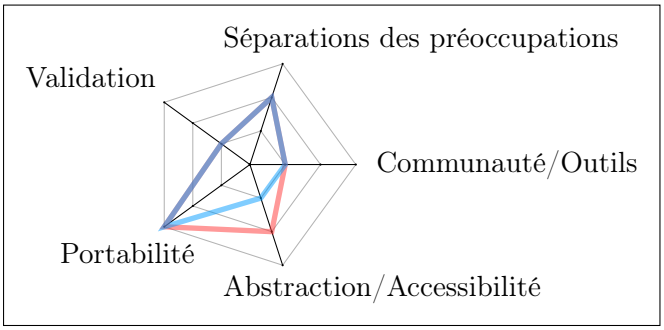
3.1.7 Support d'exécution

Les nouvelles architectures soulèvent ou complexifient plusieurs problèmes : équilibrage de charge, tolérance aux pannes, gestion de l'énergie, hétérogénéité des unités d'exécution. Ces derniers rendent de plus en plus difficile la prise de décisions concernant la répartition des calculs sur une machine au moment du développement. Face à ce constat, plusieurs projets tel que *Qilin* [144] ou *StarPU* [14] proposent de décrire les applications sous forme de tâches et de déléguer leur ordonnancement à un support d'exécution. L'idée est de pouvoir assigner à la volée les tâches à l'unité d'exécution la plus à même, en termes de performance, de réaliser le calcul.

Afin de pouvoir utiliser cette approche sur des machines hétérogènes, il est nécessaire de produire des tâches multiversionnées. Ce type de tâche nécessite une implémentation logicielle par type d'architectures. C'est ensuite le support d'exécution qui va sélectionner l'implémentation adaptée à l'architecture choisie pour l'exécution de la tâche. Le développement des différentes implémentations peut se faire, soit de façon manuelle comme c'est le cas avec *StarPU*, soit de façon automatique comme le propose *Qilin*. Ce dernier fournit une interface de programmation pour décrire les tâches. Le compilateur *Qilin* utilise les appels à cette *API* pour générer dynamiquement le code de différentes implémentations : *Intel TBB* pour les processeurs multicœurs (section 3.1.3) et *Nvidia CUDA* pour les processeurs graphiques (section 3.1.10).

En ce qui concerne l'ordonnancement des tâches, *StarPU* offre plus de souplesse que son concurrent. *Qilin* permet uniquement la soumission de tâches opérant sur des tableaux qui sont automatiquement découpés et répartis sur l'ensemble des unités d'exécution. De plus *Qilin* ne permet pas d'exécuter des tâches différentes simultanément. De son coté *StarPU* possède un ordonnancement de tâches qui n'est pas dicté par la décomposition de tableaux, les décisions étant prises au niveau des tâches en fonction du graphe des dépendances.

Légende :
- *Qilin*
- *StarPU*



3.1.8 Assemblage de composants parallèles

Le développement de logiciels de simulation numérique est un domaine où la réutilisation de code est fréquente. En effet les opportunités sont nombreuses : rénovation d'un code, développement d'un code multiphysique, partage entre différentes équipes, etc. C'est face à ce constat, que des travaux prônant l'utilisation de composants logiciels pour le calcul haute-performance sont apparus [10, 24, 37].

Common Component Architecture (CCA)

Le *Common Component Architecture* [10] est une spécification de modèle de composant visant à définir la meilleure façon d'appliquer les principes de la programmation orientée composant au domaine du calcul scientifique. *CCA* définit trois concepts fondamentaux : *component*, *port*, *framework*.

- Les *components* sont des briques logicielles fonctionnant comme des boites noires. Ils cachent leur complexité interne et n'exposent que des interfaces clairement définies.
- Les *ports* sont les interfaces abstraites au travers desquelles interagissent les *components*. Il existent deux types de *port*, ceux qui implémentent une fonctionnalité (*provides ports*) et ceux qui utilisent une fonctionnalité (*uses ports*).
- Les *frameworks* ont la charge de gérer l'assemblage et l'exécution d'applications construites à partir de *components*. Ils sont notamment responsables de connecter les *uses ports* et *provides ports* entre eux.

CCA n'étant qu'une spécification, on trouve plusieurs *frameworks* tel que *Ccaffeine* [9] et *SCIRun2* [215] qui l'implémentent.

Une des forces de l'approche *CCA* réside dans sa capacité à combiner des composants écrits dans des langages de programmation différents. Deux éléments permettent la réalisation d'un tel objectif. Tout d'abord les interfaces des composants sont décrites à l'aide d'un langage unique, le *Scientific Interface Definition Language* [80]. Ensuite, à partir de cette description, l'outil interopérabilité de langage *Babel* [78] génère le code intermédiaire permettant au fournisseur et à l'utilisateur de l'interface de communiquer.

L'inconvénient majeur de *CCA* est qu'il ne fournit rien au delà de l'assemblage des composants. Rien n'est dit sur le contenu des composants. Ils doivent donc toujours dépendre d'une solution logicielle classique tel que *MPI* pour la description de leur algorithme parallèle.

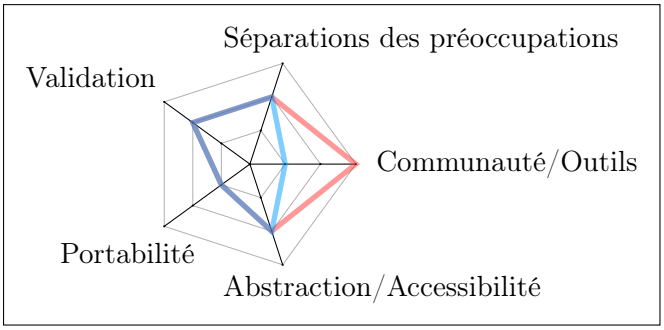
High Level Component Model (HLCM)

HLCM [37] est un modèle de composants logiciels qui a pour particularité d'accepter l'ajout d'opérateurs de composition (assemblage de composants) sans modification du modèle. Les modèles *HLCM* étant abstraits, une phase de spécialisation est nécessaire. C'est grâce à une approche basée sur l'ingénierie dirigée par les modèles, que les modèles *HLCM* sont spécialisés vers des modèles de composants existants tel que *CCA* présenté dans la section précédente. Il existe un prototype de mise en œuvre de *HLCM* appelé *HLCM/CCM*, qui s'appuie sur le modèle de composant de *CORBA* [107].

Bien qu'*HLCM* facilite une bonne structuration des applications ainsi que la réutilisation de composants, il hérite de certaines faiblesses des modèles de composants sur lesquels il s'appuie. Comme nous venons de le voir pour *CCA*, rien n'est proposé pour le développement interne des composants. Dans ces conditions, comment définir un composant capable de s'exécuter aussi bien sur une architecture de type *CPU* que sur une architecture de type *GPU*? La réutilisation et la portabilité ont donc leur limite avec cette approche.

Évaluation de la catégorie

Légende :
- *CCA*
- *HLCM*



3.1.9 Parallélisme quasi-synchrone

Le modèle de programmation parallèle quasi-synchrone, *Bulk Synchronous Parallelism* (*BSP*) en anglais, a été proposé par Valiant [206] en 1990. Un programme basé sur le modèle *BSP* se décompose en plusieurs étapes appelées *superstep* (voir figure 3.3). Chaque *superstep* comprend une phase de calcul asynchrone, une phase de communications et se termine par une phase de synchronisation forcée. Dans ce modèle, le calculateur est vu comme un ensemble homogène d'unités d'exécution reliées entre elles par un réseau de communication et possédant une unité de synchronisation globale.

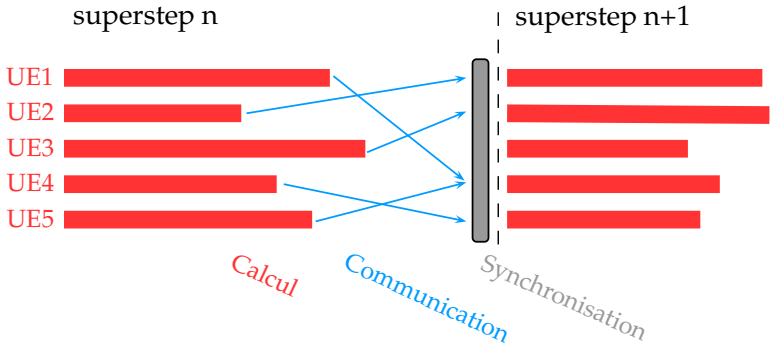


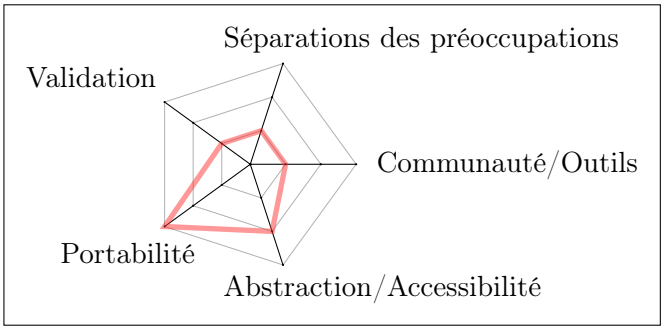
FIGURE 3.3 – Principe du modèle *Bulk Synchronous Parallelism*

L'avantage du modèle *BSP* est qu'il assure l'absence d'inter-blocage que l'on retrouve, entre autres, dans la programmation par *thread*. Il limite aussi fortement les possibilités de non-déterminisme des programmes. Du côté des inconvénients, il faut noter qu'il n'est pas de possible d'exprimer du parallélisme imbriqué et que les problèmes irréguliers (c.-à-d. un problème qui va aboutir à la création de tâches de calcul de tailles très différentes) sont difficilement traitables.

De nombreux travaux [191, 38] ont étendu le modèle initial proposé par Valiant pour pallier certains de ces problèmes. Récemment, Hamidouche et coll. ont proposé *BSP++* [115] une bibliothèque générique inspirée de l'approche fonctionnelle et data-parallèle de la bibliothèque *BSMLLib* [140] ainsi que de l'approche *BSP* hiérarchique proposée par la bibliothèque *D-BSP* [32]. En addition de cette bibliothèque, ils ont développé un *framework* de génération nommé *BSPGen* [116]. Ce dernier est capable, grâce au modèle de cout de *BSP++*, de générer le code décrivant un enchainement efficace des *supersteps* pour des machines hybrides à plusieurs niveaux en utilisant le couple *MPI*/*OpenMP* ou *MPI* avec le *SDK* du *Cell BE*. Bien que cette approche réponde à certains de nos problèmes, elle n'est pas

exempte de faiblesses. Par exemple, la bibliothèque *BSP++* repose sur le modèle *SPMD* qui, comme on l'a déjà vu dans la section 3.1.2, n'est pas le plus naturel pour le développeur.

Légende :
– *BSP++*



3.1.10 Accélérateurs matériels

Comme nous l'avons vu dans la section 2.2.2, les accélérateurs sont plébiscités. Leur performance et leur consommation électrique expliquent cet engouement. Cependant, nous allons voir dans cette section que la situation n'est pas aussi idyllique du côté des solutions logicielles permettant de les exploiter.

Compute Unified Device Architecture (CUDA)

Le langage *CUDA* [132, 183] est une extension du langage *C* qui est développé par le constructeur de cartes graphiques *NVIDIA* afin d'exploiter la puissance de calcul de ses *GPU*. Le langage *CUDA* permet d'exprimer le parallélisme de données d'une application via l'utilisation d'un grand nombre (de l'ordre de la centaine) de processus très légers.

Dans le modèle de programmation *CUDA*, on retrouve le concept d'*Host* qui fait référence au *CPU* et le concept de *Device* qui fait référence au *GPU*. Le langage *CUDA* permet de décrire des fonctions (tâches élémentaires) appelées *kernels* pouvant être exécutées par des *devices*. C'est l'*host* qui va déléguer l'exécution des *kernels* vers les différents *devices* disponibles sur le système. Une fois que les données ont été transférées vers le *device* concerné, le *kernel* va être exécuté par un groupe de processus. Il existe une hiérarchie dans les processus et leur mémoire qui découle directement de l'architecture matérielle (voir section 2.2.2 et notamment la figure 2.15). Les processus sont regroupés en bloc (*thread blocks* qui sont alignés sur les *streaming multiprocessors*) au sein desquels ils peuvent se synchroniser et communiquer par le biais d'une mémoire partagée. Au final on retrouve trois niveaux de mémoire à gérer : processus, bloc et *device*.

Le langage *CUDA* possède donc un modèle de bas niveau qui demande beaucoup d'efforts de la part du programmeur pour obtenir d'excellentes performances. Cette faiblesse ne l'a cependant pas empêché de connaitre un certain engouement au cours des dernières années [2]. Bien que le langage *CUDA* soit spécialement développé pour les *GPU NVIDIA*, des projets de recherche tentent d'exécuter du code *CUDA* sur d'autres architectures [169]. Il existe entre autre le projet *Ocelot* [67] qui vise les *GPU* du constructeur *AMD* et les processeurs de type *x86*.

OpenCL

Le standard *OpenCL* [131, 196], promu par le consortium *Khronos Group*, se compose d'un langage portable de bas niveau dérivé du langage *C* permettant d'écrire des noyaux de calcul pour accélérateurs (*kernels*) ainsi que d'une interface de programmation permettant

de gérer le lancement des *kernels* sur les accélérateurs. Même si le standard *OpenCL* partage une partie de son modèle de programmation avec le langage propriétaire *CUDA*, ses objectifs sont tout autres. En effet le standard *OpenCL* vise la portabilité. Par conséquent, les *devices* cibles peuvent être de nature très différente : *GPU* de différents constructeurs (*AMD*, *NVIDIA*, *VIA*), processeurs multicœurs, processeurs *Cell BE*, etc.

Malheureusement la promotion d'une interface standard de bas niveau possède son lot d'inconvénients. Tout d'abord, elle empêche les constructeurs d'exposer les caractéristiques avancées d'une architecture donnée ; c'est pourquoi il est parfois possible d'atteindre de meilleures performances sur les *GPU Nvidia* en utilisant *CUDA* plutôt qu'*OpenCL* [76]. Ensuite, dans la philosophie d'*OpenCL*, un *kernel* peut être exécuté sur des *devices* ayant des architectures matérielles différentes mais la portabilité des performances n'est pas garantie. Au final le développeur devra spécialiser le code d'un *kernel* pour une architecture donnée s'il souhaite obtenir d'excellentes performances avec ce dernier.

Hybrid Multicore Parallel Programming (HMPP)

La solution *HMPP* [71, 40] propose une approche pragmatique qui repose sur l'annotation de code source existant en *C*, *C++* et *Fortran* via des directives de compilation comme le proposait déjà *OpenMP* (voir section 3.1.3). Mais, à la différence d'*OpenMP* qui ne cible que les *CPU*, *HMPP* est capable d'exploiter les accélérateurs. En effet la solution *HMPP* comprend un compilateur *source-to-source* capable de générer du code *CUDA* ou *OpenCL* à partir des informations fournies par les directives de compilation (figure 3.4).

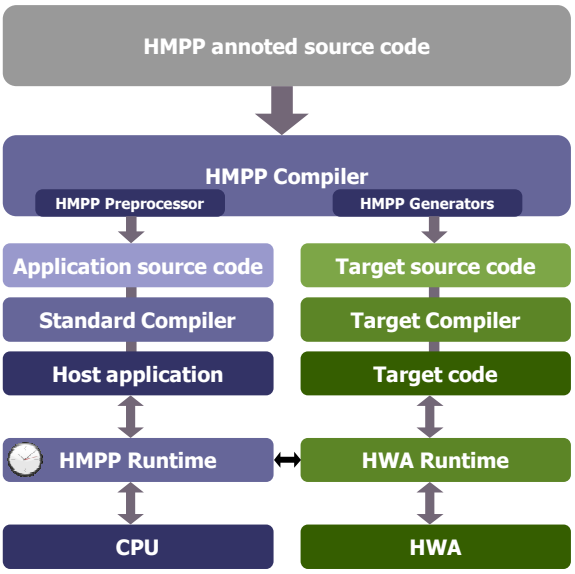
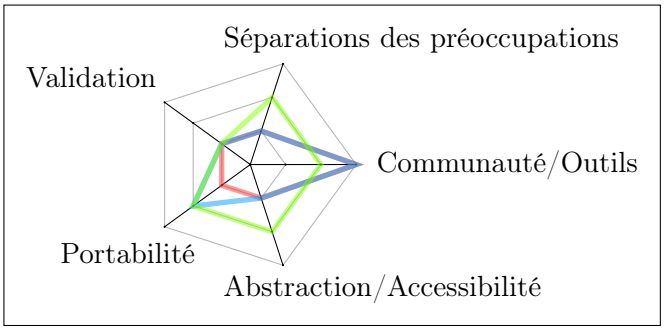


FIGURE 3.4 – Processus de compilation et d'exécution du *workbench HMPP*[40]

Les directives *HMPP* permettent d'identifier des fonctions, appelées *codelets*, dont on souhaite déléguer l'exécution aux accélérateurs. Elles permettent aussi de spécifier les transferts de données de même que les conditions d'exécution de ces fonctions (synchrone, asynchrone, garde).

Évaluation de la catégorie

Légende :
- *CUDA*
- *OpenCL*
- *HMPP*



3.1.11 Langages dédiés

Les solutions logicielles présentées jusque là ont vocation à être généralistes, c'est-à-dire qu'elles permettent de développer des applications parallèles pour des domaines variés : physique, biologie, informatique, finance, etc. À l'opposé, les langages dédiés, appelés aussi *Domain Specific Langage* (*DSL*), se restreignent à domaine particulier. Ce qui de prime abord apparait comme une limitation, se révèle être un grand avantage : le développeur manipule des concepts directement issus de son domaine métier. Ces concepts métiers vont lui permettre d'exprimer son objectif : le « quoi », plutôt que de décrire le « comment » comme c'est souvent le cas avec les langages généralistes. Le nombre de langages dédiés appliqués à des sous-domaines plus ou moins grands de la simulation numérique est en constante augmentation [98, 66, 181, 125, 148]. Dans cette section, nous nous contenterons de présenter deux solutions logicielles représentatives : *Liszt* et *TCE*.

Liszt

Le langage *Liszt* [65] est un langage dédié à la simulation numérique et notamment à la résolution d'équations aux dérivées partielles (*EDP*). L'idée fondatrice de *Liszt* est de pouvoir décrire une application au niveau d'abstraction de son schéma numérique plutôt qu'à celui d'un langage parallèle explicite. L'objectif est donc conforme aux principes des *DSL* : séparer la spécification d'un problème d'une implémentation particulière.

La syntaxe du langage *Liszt* est basée sur un sous ensemble du langage *Scala* augmenté des concepts métiers propre à la résolution d'*EDP*. En effet, le langage *Scala* fournit des mécanismes d'extension et de restriction qui sont particulièrement adaptés à la création de *DSL* [179]. La notion de maillage est placée au cœur du langage *Liszt*. Sa syntaxe permet la spécification d'algorithmes manipulant les différents éléments d'un maillage (nœud, arête, face, cellule) au travers des relations les unissant. L'expression du parallélisme est possible via l'utilisation d'une construction, appelée *for-comprehension*, qui exprime un traitement à appliquer à un ensemble d'éléments issus d'un maillage. La sémantique du *for-comprehension* définit que le traitement peut être réalisé indépendamment sur chaque élément de l'ensemble. Il est possible d'exprimer plusieurs niveaux de parallélisme en imbriquant plusieurs *for-comprehension*.

Le processus de compilation du langage *Liszt* est présenté dans la figure 3.5. Grâce aux restrictions du langage apportées par l'utilisation forcée de concepts métiers et de certaines constructions, le compilateur est capable de déterminer les dépendances de données de n'importe quelle expression. En se basant sur cette information, le compilateur est alors en mesure de générer des implémentations pour plusieurs plateformes cibles. Ces implémentations sont basées sur *CUDA* pour les *GPU*, *PThread* (section 3.1.3) pour les pour

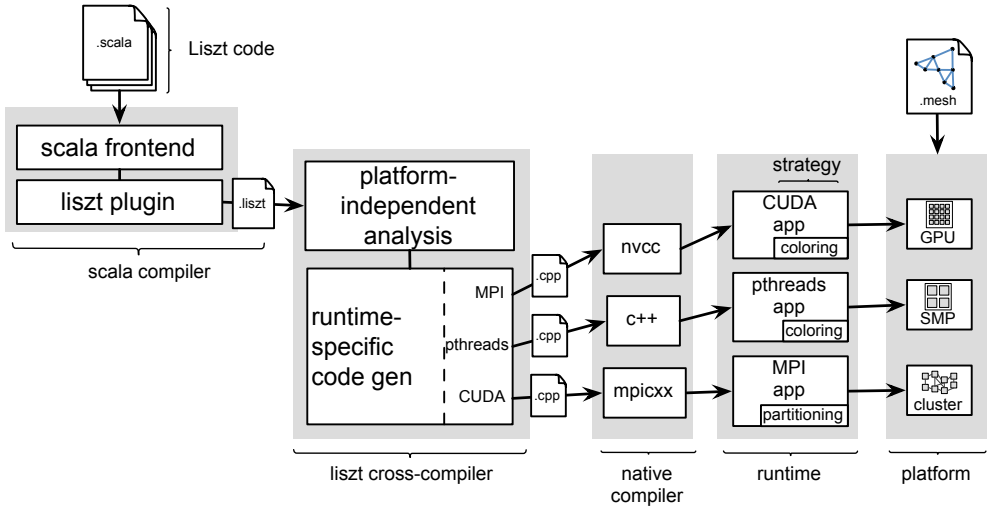


FIGURE 3.5 – Processus de compilation du langage *Liszt* [65]

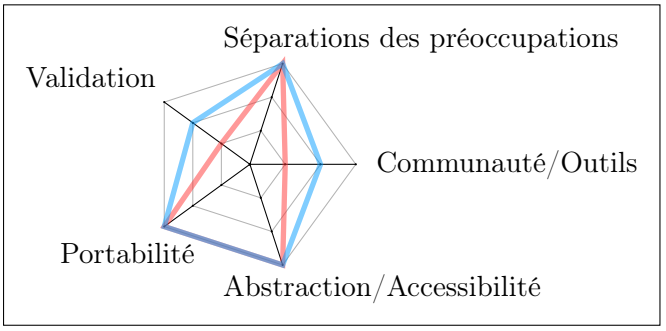
les architectures à mémoire partagée et finalement *MPI* pour les architectures à mémoire distribuée.

Tensor Contraction Engine (TCE)

TCE [26] est un langage dédié permettant de décrire des contractions tensorielles pour la chimie numérique. Il a permis de réduire les temps de développement de ce type de code de simulation de plusieurs mois à quelques heures. Ce gain important de productivité provient de la capacité de *TCE* à générer des applications massivement parallèles à partir de spécifications de haut niveau. En effet, ce dernier est capable de générer, grâce à l'enchainement de plusieurs transformations, du code utilisant le standard *MPI* ou la bibliothèque *Global Array* [163] qui adopte une approche *PGAS*.

Évaluation de la catégorie

Légende :
- *Liszt*
- *TCE*



3.1.12 Frameworks de développement

Les *frameworks* de développement possèdent des similitudes avec les langages dédiés (section 3.1.11) puisqu'ils sont aussi adaptés à une classe spécifique de problèmes. Par contre, ils diffèrent sur la façon de mettre en place cette spécificité. En effet, alors que dans le cas des langages dédiées les concepts métiers se trouvent au cœur du langage, dans le cas des *frameworks* c'est au travers d'une plateforme logicielle réutilisable bâtie sur un langage

généraliste que la connaissance métier est apportée. Ces plateformes embarquent généralement toute une collection d'artefacts logiciels : compilateurs, bibliothèques, interface de programmation, fichiers de configurations, outils, etc. Dans la suite de cette section, nous présentons deux *framework* spécifiques : *DUNE* et *Arcane*.

DUNE

Le *framework DUNE* [23] permet de concevoir des logiciels pour la résolution d'équations aux dérivées partielles via des méthodes basées sur des maillages. Il supporte l'implémentation des méthodes de discrétisation les plus courantes : éléments finis, volumes finis, différences finies.

Le projet *DUNE* se fonde sur un certain nombre de principes fondamentaux. Le premier concerne la nécessité de séparer les algorithmes et les structures de données en utilisant des interfaces abstraites. Le deuxième concerne l'implémentation performante de ces interfaces à l'aide des techniques de la programmation générique telle qu'on la trouve en *C++* [15]. Le troisième a trait à la réutilisation de bibliothèques existantes pour les implémentations concrètes des interfaces.

La plateforme *DUNE* se décompose en plusieurs modules traitant chacun d'un aspect particulier d'une simulation numérique. Les plus courants sont *dune-common* (classes de base, gestion des matrices et vecteurs), *dune-grid* (interface abstraite et implémentation pour la gestion de maillages), *dune-istl* (interface et ensemble de solveurs d'algèbre linéaire). C'est le développeur qui a la charge de déterminer l'ensemble des modules nécessaires à son application.

Arcane

Arcane [106] est une plateforme de développement logiciel écrite en *C++* pour les codes de simulation numérique basés sur des maillages 2D et 3D. Initialement développée et utilisée par le *CEA/DAM* à partir l'année 2000, la plateforme est depuis 2007 partagée dans le cadre d'une collaboration avec l'IFPEN (IFP Énergies Nouvelles) [101].

L'objectif d'*Arcane* est de gérer les aspects informatiques d'un code de simulation numérique (gestion du maillage et de la mémoire, entrées/sorties, parallélisme) afin de faciliter le travail des développeurs qui sont le plus souvent des numériciens ou des physiciens. La plateforme embarque aussi une collection d'outils (compilation, débogage, vérification et validation) ayant pour but d'augmenter la productivité des développeurs et la qualité des codes.

```
1 ENUMERATE_FACE(face, cell.faces())
2 {
3         Real3 center(0.0,0.0,0.0);
4         ENUMERATE_NODE(node, face.nodes())
5         {
6                 center += node_coor[node];
7         }
8         center /= face.nbNode();
9 }
```

Listing 3.3 – Exemple de programme *Arcane* permettant de calculer le centre d'une maille quelque soit sa dimension

Dans le même esprit que la plateforme *DUNE* (section 3.1.12), *Arcane* prône une séparation entre algorithmique et structures de données. C'est pour cette raison que la manipulation des entités d'un maillage se fait via l'utilisation d'interfaces et d'itérateurs génériques.

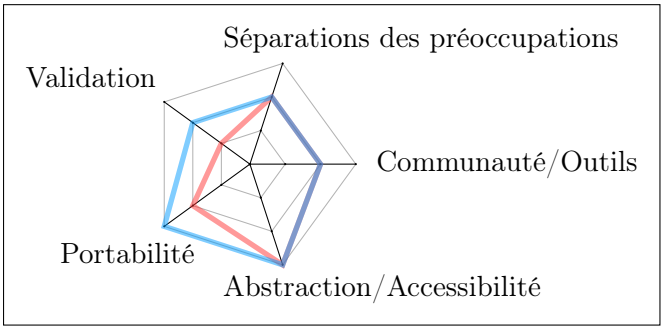
Ces interfaces ont été réalisées de telle sorte que certains codes peuvent être utilisés sur des maillages 2D ou 3D sans modification. Le listing 3.3 illustre ces principes.

Arcane permet l'expression du parallélisme à deux niveaux. Premièrement, le domaine de calcul (le maillage) est décomposé en sous-domaines. L'utilisation d'interfaces pour manipuler les éléments du maillage et d'itérateurs génériques pour décrire les algorithmes rende quasiment immédiate l'expression de ce premier niveau de parallélisme. En effet, le développeur a juste besoin de spécifier quelques informations additionnelles telle que la nécessité de synchroniser une variable entre les différents sous-domaines. Les aspects sous-jacents tels que le partitionnement ou la gestion des mailles fantômes sont rendus abstraits grâce à la plateforme. Deuxièmement, *Arcane* permet l'expression de parallélisme de tâche à l'aide de boucles parallèles.

La plateforme impose via des fichiers de description XML [42] une structuration particulière des programmes. Ces derniers sont décomposés en *modules* représentant un domaine de la physique intervenant dans le phénomène simulé. Un *module* contient généralement un ensemble de *variables* représentant des paramètres physiques, des méthodes appelées *point d'entrée* qui permettent de faire évoluer ces valeurs et la description du jeu de données du *module*. Il existe un type particulier de *modules* appelé *service* qui peuvent être utilisés par les *modules* traditionnels. Une application de simulation numérique est décrite à l'aide d'un autre type de fichier XML qui va spécifier la liste des *modules* utilisés ainsi que la séquence d'exécution des *points d'entrée* dans une boucle appelée boucle en temps. Elle est appelée ainsi car une itération de cette boucle représente l'évolution du phénomène physique modélisé pendant la durée du pas de temps courant.

Évaluation de la catégorie

Légende :
- *DUNE*
- *Arcane*



3.1.13 Basé sur les modèles

La modélisation d'applications parallèles est loin d'être une idée nouvelle. Des projets tel que *GRAPNEL* [77], *CODE 2.0* [162] ou encore *Hence* [28] ont ouvert la voie dans les années 1990. Ces travaux s'inspirent du paradigme de communication par passage de messages (section 3.1.2). Ils combinent le plus souvent une syntaxe graphique (graphe orienté où les nœuds représentent des calculs à effectuer et les arcs représentent les dépendances de données) avec une syntaxe textuelle pour la spécification des calculs. La figure 3.6 illustre ce principe au travers d'un exemple de graphe *CODE 2.0*. Ces environnements proposaient déjà des outils de génération de code plus ou moins évolués. Malheureusement, le niveau d'abstraction de ces langages n'était pas beaucoup plus élevé que celui des langages qu'ils pouvaient générer.

C'est au début des années 2000 que Pllana et coll. [175] proposèrent d'utiliser le standard UML pour modéliser les applications de calcul haute-performance. Les modèles pro-

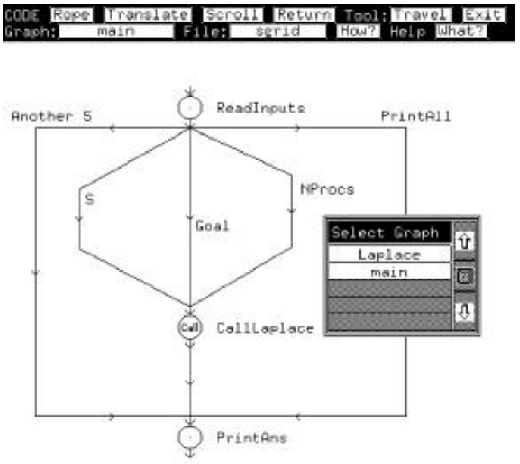


FIGURE 3.6 – Exemple de graphe *CODE 2.0* [162]

duits servent, non pas à générer le code des applications parallèles correspondantes, mais à prédire les performances [173]. Plus récemment, les travaux de Pllana et coll. ont évolué au-delà de la phase de spécification et adoptent les principes de l'ingénierie dirigée par les modèles (*IDM*). Ils proposent de créer un environnement de programmation intelligent pour le développement d'applications visant les architectures multicœurs [174]. Cet environnement repose sur la combinaison de l'*IDM*, d'agents logiciels intelligents [43] et sur l'utilisation de *skeletons* parallèles appelés *parallel building blocks*.

Dans la suite de cette section, nous nous concentrerons sur les projets récents les plus en adéquation vis à vis de notre problématique que sont *Syntony* et *Gaspard*.

Syntony

Syntony [68] est initialement une plateforme de transformation de modèles dédiée à la génération de logiciels de simulation à événements discrets [18]. En partant des modèles *UML* adéquats (*class diagram*, *composite structure diagram*, *state machine diagram*, *activity diagram*), elle est capable de générer automatiquement l'application correspondante. Le code source produit est en *C++* et s'appuie sur la bibliothèque *OMNeT++* [209] qui fournit un moteur d'exécution de simulation à événements discrets.

Dans des travaux plus récents [69], Dietrich et coll. ont étendu *Syntony* afin de prendre en compte les applications de simulation continue. Le processus de développement proposé est présenté dans la figure 3.7. Dans l'exemple publié, ils partent d'un *framework* multigrille *C++* pour le traitement d'images par approches variationnelles [136]. En utilisant les principes de la rétro-ingénierie, ils récupèrent les modèles *UML* correspondants à ce *framework*. Un développeur peut ensuite modéliser une application via des diagrammes de classes et d'activités *UML* qui vont combiner des briques logicielles du *framework*. Ce sont ces briques logicielles qui fournissent la possibilité d'exploiter des architectures matérielles comme les *GPU* ou *Cells* (section 2.2.2). Une fois l'application modélisée, la plateforme *Syntony* peut générer l'application finale en embarquant seulement les éléments utiles du *framework*.

Un des points forts de l'approche est certainement la définition d'un atelier de génie logiciel complet, implémentée actuellement avec la plateforme *Eclipse* [55], qui couvre l'ensemble du cycle de développement et d'utilisation des applications. Du côté des points faibles, en écartant le problème du couplage extrêmement fort qui existe entre le modèle d'application et le modèle du framework, l'inconvénient majeur de cette approche est que l'expression et la gestion du parallélisme sont entièrement déléguées au *framework*. Le pro-

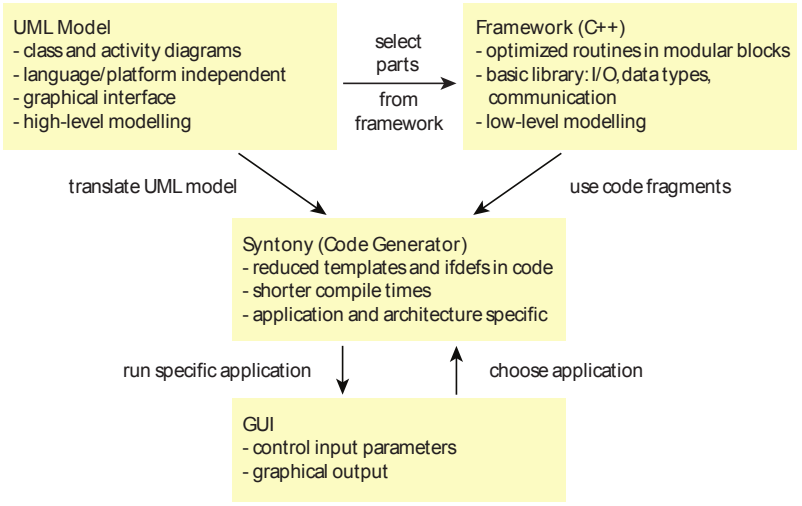


FIGURE 3.7 – Processus de développement de l'approche *Syntony* [69]

blème du développement d'applications parallèles est donc toujours présent, il a juste été reporté sur le *framework*.

Gaspard

Ce sont certainement les travaux de l'équipe *DaRT*[1] qui se rapprochent le plus des travaux présentés dans cette thèse. Bien que spécialiste des systèmes embarqués, des travaux portant sur la simulation numérique ont été réalisés. L'équipe a proposé une extension de leur méthodologie de développement d'applications pour les systèmes embarqués afin de couvrir le développement d'applications de simulation numérique. Ils se sont basés sur les principes de l'*IDM*. La spécification des applications est réalisée grâce à un profil *UML*. Cette spécification est ensuite utilisée par une chaine de transformations pour produire le code de l'application. Cette approche est implémentée au sein de l'environnement *Gaspard 2* qui est présenté dans la figure 3.8.

Les premiers travaux [201] utilisaient le profil *UML Gaspard* [29] avec une chaine de transformations permettant de générer du code *OpenMP*. Par la suite, l'équipe a participé à la définition d'un profil standardisé par l'*OMG* : *Modeling and Analysis for Real-Time and Embedded systems* (*MARTE*) [96]. Comme son nom l'indique, *MARTE* est un profil *UML* dédié à la spécification d'applications temps-réel et embarquées. Il partage de nombreux concepts avec le profil *Gaspard*.

Leurs derniers travaux [178] se basent donc naturellement sur le profil *UML MARTE* et proposent une nouvelle chaine de transformations permettant de générer du code *OpenCL* (section 3.1.10) pouvant s'exécuter sur des *GPU*.

Que ce soit dans le cas du profil *Gaspard* ou dans celui de *MARTE*, l'expression du parallélisme est inspirée d'*Array Oriented-Language* (*Array-OL*) [64]. *Array-OL* est un langage de spécification de haut niveau manipulant des tableaux. Il est adapté à la définition d'applications de traitement du signal intensif. La spécification d'une application utilisant ce formalisme repose sur deux modèles :

– un « modèle global » qui permet de décrire l'ensemble de la chaine de traitement sous forme d'un graphe de tâches dirigé acyclique. Les nœuds représentent les tâches et les liens représentent les échanges de données (tableaux) entres les tâches.

1. http://team.inria.fr/dart/

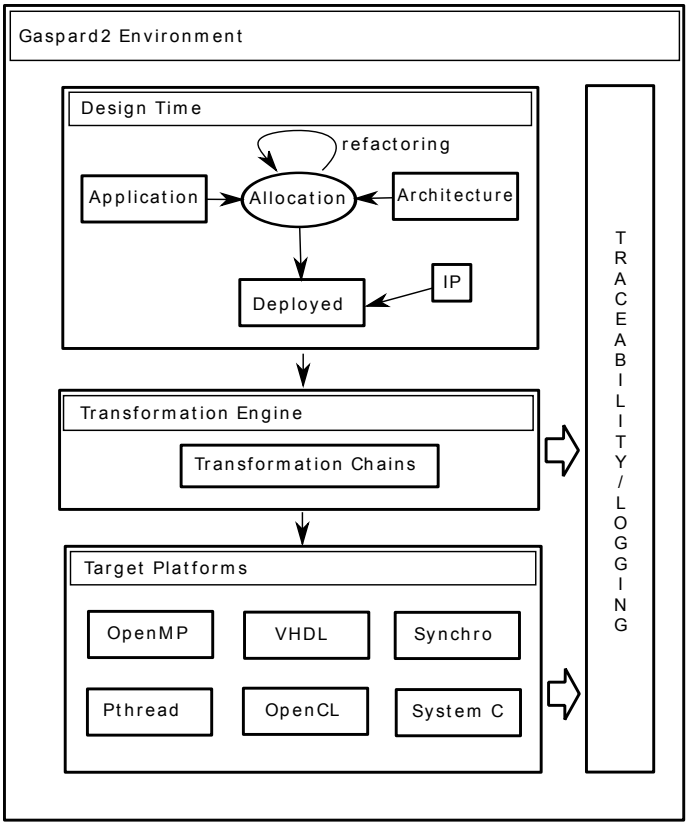


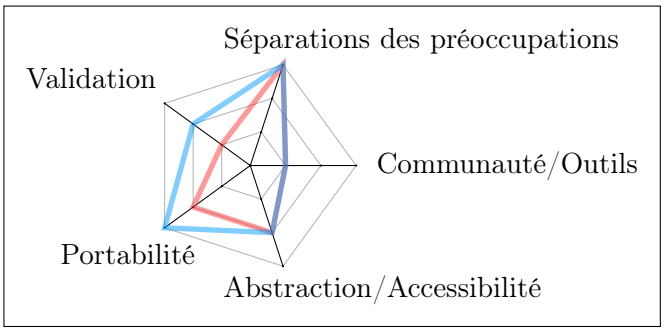
FIGURE 3.8 – Environnement de développement *Gaspard 2* [63]

– un « modèle local » qui permet de définir les calculs et accès aux tableaux pour chaque tâche.

Ces travaux proposent des avancées en termes de séparation des préoccupations et de portabilité. Par contre le principal reproche que l'on peut faire à cette approche concerne son accessibilité. En effet, les profils *Gaspard* et *MARTE* ont été conçus pour modéliser des applications temps-réel et embarquées. Par conséquent, ils ne proposent seulement qu'une partie des concepts nécessaires pour modéliser des applications de calcul scientifique. Á l'inverse, ils complexifient le processus de modélisation en proposant des concepts qui n'ont aucune utilité dans ce cadre.

Évaluation de la catégorie

Légende :
– *Syntony*
– *Gaspard*



3.1.14 Discussion

Des différences de niveau d'abstraction apparaissent clairement dans le panel de solutions logicielles que nous venons de présenter (figure 3.9). D'ailleurs on remarque que de nombreuses solutions de haut niveau d'abstraction comme *Liszt* utilisent des solutions de bas niveau plutôt que de développer une pile logicielle complète permettant d'exploiter une architecture parallèle donnée. Cette possibilité, facilitée par la montée en abstraction, permet de réduire le coût nécessaire à la prise en compte de nouvelles plateformes d'exécution.

La figure 3.9 permet de constater que c'est la propriété de validation qui fait le plus souvent défaut dans les solutions logicielles actuelles. Or, la validation joue un rôle crucial puisqu'elle est nécessaire pour quantifier la crédibilité que l'on peut accorder aux résultats d'une simulation numérique.

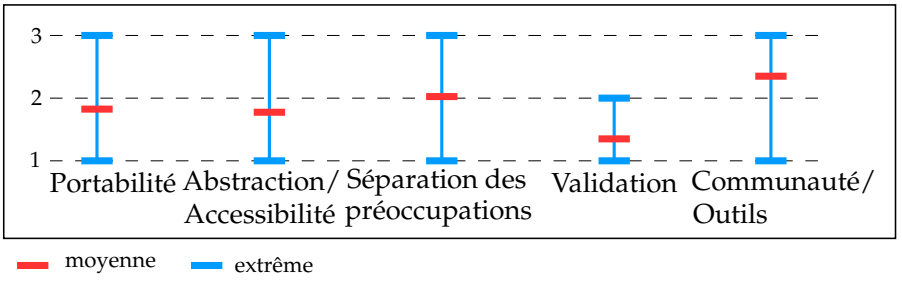


FIGURE 3.9 – Moyennes et extrêmes de la complétude des propriétés

Bien que l'on ait volontairement délaissé le critère de la performance dans cette évaluation (méthodologie en annexe A), il faut garder à l'esprit que la possibilité d'exécuter une application sur une architecture donnée ne garantit en aucun cas que cette dernière sera exploitée efficacement. D'ailleurs, c'est cette quête de performances qui pousse les développeurs à combiner des solutions logicielles de bas niveau pour utiliser le potentiel des machines. Il existe des développements utilisant directement des couples tel que *MPI/OpenMP*, *MPI/HMPP*,*MPI/CUDA* voire même *MPI/OpenMP/CUDA* sans génération de code de la part d'un outil de plus haut niveau. Par exemple, dans le cas d'un *cluster* avec des nœuds *ccNUMA* (section 2.2.1), le couple *MPI/OpenMP* peut être choisi. *MPI* est utilisé pour la communication entre nœuds et *OpenMP* est en charge d'exploiter la puissance de calcul des nœuds. Ce type d'approche n'apporte pas forcément les gains de performances escomptés [47, 119, 75]. Deux inconvénients viennent s'ajouter à ce premier constat : d'une part, la complexité de programmation augmente à cause du mélange des paradigmes et, d'autre part, la portabilité des applications diminue à cause de leur spécialisation.

3.2 Génie logiciel et calcul scientifique

L'utilisation de méthodes et techniques issues du génie logiciel n'est pas une pratique courante dans le domaine du calcul scientifique [48, 22, 186]. On peut citer deux raisons expliquant ce manque. Premièrement, la formation initiale des développeurs de code de simulation numérique est traditionnellement une formation en physique ou en mathématiques appliquées avec peu, voire pas, de connaissances en génie logiciel à la sortie de cette dernière. Deuxièmement, l'expression du besoin ne peut être réalisée de façon classique. Le besoin se résume parfois à un modèle physique et peut évoluer très fréquemment au cours du développement en fonction des avancées scientifiques. Une approche de type « *big design*

up-front » comme dans un processus de développement utilisant un cycle en cascade n'est donc pas adaptée.

Il ne faut pas perdre de vue qu'un logiciel de simulation n'est qu'un moyen utilisé par les scientifiques pour valider ou utiliser des modèles physiques : les contraintes ne sont donc pas les mêmes que pour un logiciel commercial plus « traditionnel ». Pendant longtemps, ces différences n'ont intéressé que faiblement la communauté du génie logiciel. Cependant, la situation évolue, et plus particulièrement depuis la dernière décennie qui recense un certain nombre de travaux sur cette thématique que nous abordons dans cette section.

3.2.1 Conception guidée d'applications parallèles

Cette section s'attache à présenter deux méthodes de conception disponibles pour le développement d'applications parallèles : les patrons de conception et les squelettes algorithmiques.

Patrons de conception

La notion de patron de conception (*design pattern* en anglais) a été introduite par Alexander et coll. dans un traité d'architecture et d'urbanisme [8] datant de 1977. Un *design pattern* peut être défini comme « une solution à un problème récurrent ». La solution se présente sous la forme d'une règle de conception issue de l'expérience qui permet de satisfaire les contraintes liées au problème. C'est une forme de capitalisation du savoir. Il faudra attendre 1987 pour que Beck et coll. proposent d'appliquer le principe à la conception de logiciels [27].

Des travaux sur les *design pattern* parallèles apparaissent au début des années 2000. Mac Donald et coll. proposèrent, au sein de l'approche CO_2P_3S (*Correct Object-Oriented Pattern-based Parallel Programming System*) [146], la notion de « *generative parallel design pattern* ». Contrairement à un patron de conception classique qui est un artefact passif, le patron génératif produit une partie d'application. L'idée est la suivant : le développeur sélectionne un patron de conception parallèle et le paramétrise pour son problème. L'outil de génération embarqué dans CO_2P_3S va alors générer le code structurel de l'application [145]. Par la suite, le développeur n'a plus qu'à implémenter l'algorithmique particulière à son application. L'outil de génération est capable de générer du code parallèle pour les architectures à mémoire distribuée en utilisant le langage *Java* et la bibliothèque *RMI* [202]. Le processus global de développement promu par CO_2P_3S est présenté dans la figure 3.10.

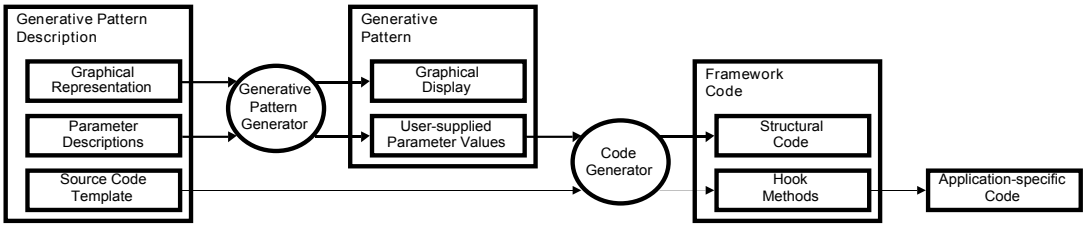


FIGURE 3.10 – Approche CO_2P_3S par patron de conception génératif [145]

À la même période, Massingill et coll proposèrent un ensemble de patrons de conception parallèles (*PLPP*) accompagné d'une méthodologie pour les utiliser [147, 149]. Les patrons sont répartis entre quatre catégories appelées *design space* et représentant chacune une étape du développement :

– *Finding Concurrency* : cette catégorie explore les façons de structurer le problème pour identifier les sources de parallélisme.

– *Algorithm Structure* : elle définit comment structurer l'algorithme pour exploiter le parallélisme identifié à l'étape précédente. On trouve une organisation basée, soit sur des tâches (*Task Parallelism*, *Divide and Conquer*), soit sur la décomposition de données (*Geometric Decomposition*, *Recursive Data*) ou sur un flot de données (*Pipeline*, *Event-Based Coordination*).
– *Supporting Structure* : elle offre plusieurs modèles permettant de structurer les programmes (*SPMD*, *Master/Slave*, *Loop parallelism*, *Fork/Join*) ou les structures de données (*Shared Data*, *Shared Queue*, *Distributed Array*).
– *Implementation Mechanisms* : cette catégorie concerne les problèmes liés à l'implémentation de bas niveau : gestion des unités d'exécution, synchronisation et gestion des communications.

Ces travaux ont été étendus par une équipe du *ParLab*[2] de l'université de Berkeley afin de prendre en compte l'architecture logicielle complète d'une application parallèle [129]. Récemment, les travaux sur *PLPP* et ceux du *ParLab* ont été fusionnées afin de créer un collection plus large de patrons de conception [130]. La figure 3.11 donne un aperçu de la structure d'*OPL* (*Our Pattern Language*), le langage de patrons de conception parallèle issu de cette réunion.

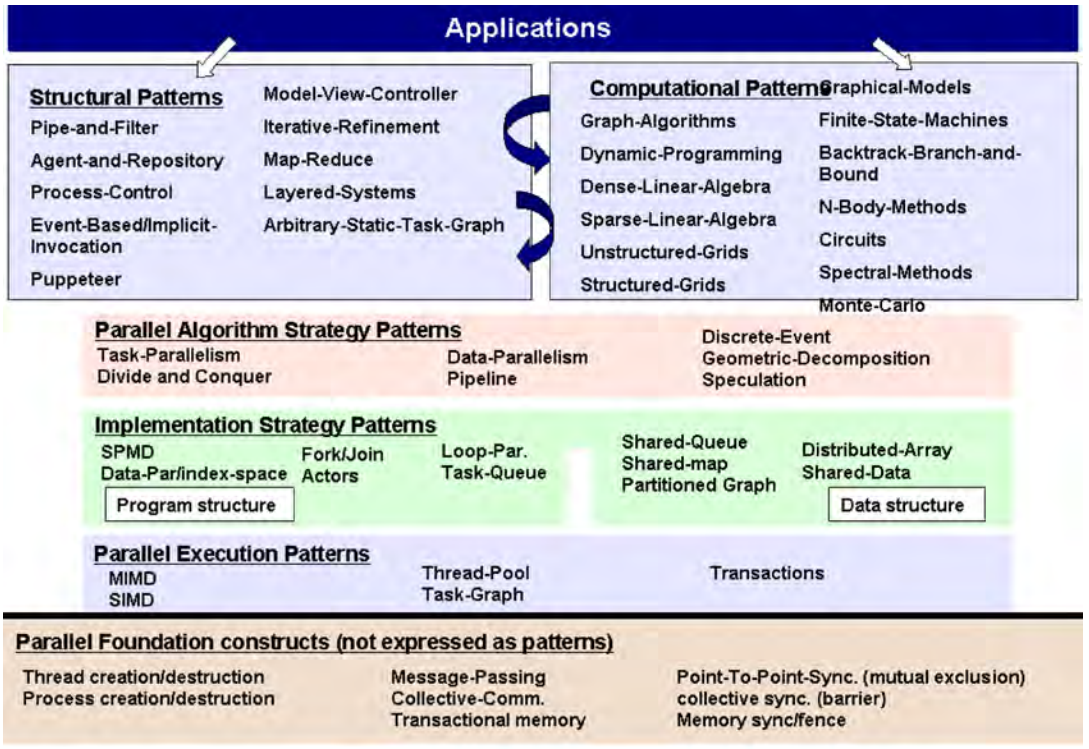


FIGURE 3.11 – Structure de *Our Pattern Language* [130]

Squelettes algorithmiques

Le concept de squelette algorithmique a été défini par Cole lors de son doctorat à la fin des années 80 [57] :

2. http://parlab.eecs.berkeley.edu/

> *« An algorithmic skeleton describes the structure of a particular style of algorithm, in the way in which higher order functions represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton. The implementation task is simplified by the fact that each skeleton may be considered independently, in contrast to the monolithic programming interfaces of existing systems at a similar level of abstraction. »*
>
> M. Cole [57]

L'idée des squelettes algorithmiques est donc de proposer des structures de programmation paramétrables contenant des stratégies parallèles communément rencontrées. Ils peuvent être aussi perçus comme des fonctions d'ordre supérieur dont la sémantique parallèle serait implicite. Les squelettes algorithmiques permettent donc de cacher les détails d'implémentation des stratégies parallèles qu'ils contiennent. L'approche permet l'accès à un haut niveau de performance au plus grand nombre puisque l'implémentation du parallélisme est déléguée aux développeurs experts. En ce qui concerne le portage d'un ensemble d'applications sur une nouvelle machine, seuls ces experts interviennent dans le processus en réécrivant l'implémentation parallèle des squelettes algorithmiques.

Les recherches sur le sujet sont très nombreuses et actives mais, bien qu'il existe plusieurs dizaines de projet basés sur les squelettes algorithmes [100], leur utilisation n'est pas répandue en dehors du monde académique [58].

Comparaison

On constate globalement que l'approche par patrons de conception et celle par squelettes algorithmiques partagent la même philosophie. Cependant ils diffèrent sur plusieurs points que sont l'expressivité, l'implémentation et les possibilités d'évolution. L'approche par patrons de conception est plus axée sur la réutilisation du processus de conception alors que les squelettes algorithmiques permettent la réutilisation de code. D'ailleurs, les squelettes algorithmiques ne fournissent un cadre au développement que pour un niveau de conception déjà détaillé. Du côté de la maintenance adaptative, les squelettes algorithmiques sont préférables car il possible de reporter la migration sur ces derniers.

3.2.2 Cycle de développement

Kepner [128] dresse une liste des cycles de vie sous forme canonique typiquement rencontrés pour des logiciels de calcul haute-performance (voir schémas de la figure 3.12). Par exemple, les développements logiciels au sein du *CEA/DAM* se rapprochent des développement de type *Enterprise*. De prime abord, ces cycles de vie paraissent assez traditionnels, cependant en regardant dans le détail, on découvre des éléments propres aux codes de simulation haute-performance : action de portage du code sur un nouveau calculateur (*port*), tests et validation dans une configuration massivement parallèle (*scale*), optimisation pour la performance (*optimize*).

Il n'existe pas de cycle de développement conçu spécifiquement pour s'adapter aux particularités du calcul haute-performance. Toutefois, on constate généralement que les cycles de développement itératif ou semi-itératif de type agile sont plus adaptés que les cycles de type cascade ou en V [22].

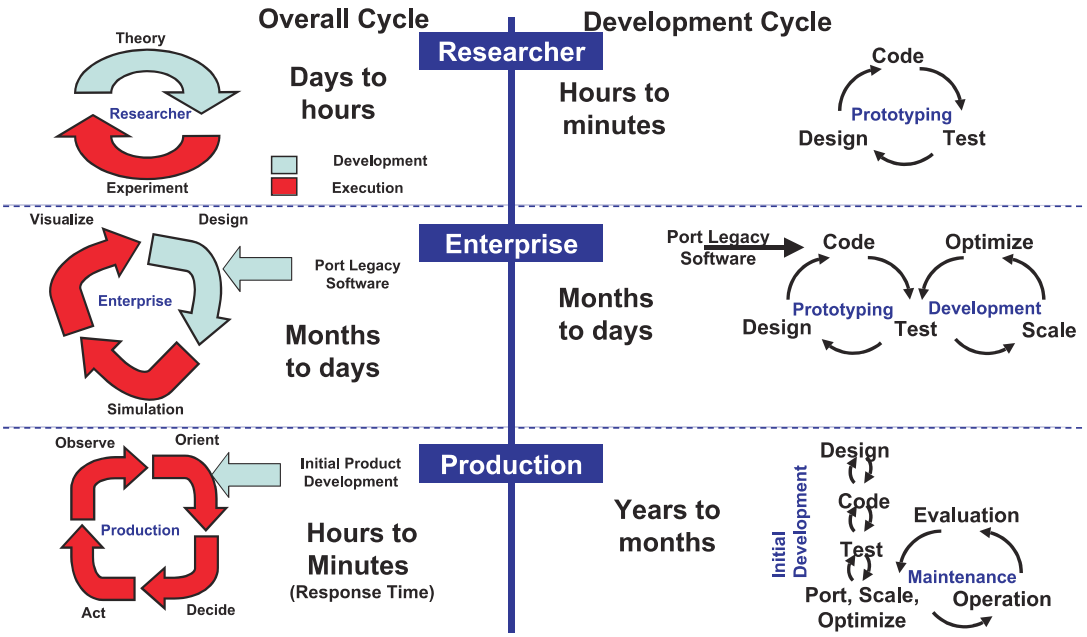


FIGURE 3.12 – Cycles de vie sous forme canonique typiquement rencontrés pour les logiciels de calcul haute-performance [128]

3.3 Conclusion

Des centaines de langages de programmation ont été créés au cours des dernières décennies. Beaucoup d'entre eux proposent des avancées techniques et des concepts de plus haut niveau qui facilitent l'expression d'algorithmes parallèles. Pourtant, ces langages ont du mal à s'installer dans la communauté du calcul scientifique. Quelques standards, dont en particulier *MPI*, dominent toujours le marché malgré leurs lacunes. Le succès et la pérennité d'un langage ne sont donc pas simplement liés à sa qualité intrinsèque. Face à ce constat, on ne peut pas considérer le couple « application de simulation numérique / solution logicielle » comme une relation stable pour une durée de vie de 20 à 30 ans.

Nous pensons qu'il manque une approche de conception dédiée à la simulation numérique qui interviendrait très tôt dans le cycle de vie d'un logiciel de calcul scientifique. C'est pourquoi dans les chapitres suivants, nous proposerons un paradigme de conception basée sur l'*IDM* et inspirée des composants, des patrons de conception et des squelettes algorithmiques. Cette approche ne vient pas concurrencer les solutions logicielles que nous venons de présenter. Elle se situe au dessus de ces dernières afin d'être en mesure de tirer partie des qualités de chacune d'entre elles. Nous estimons aussi que la description algorithmique des composants de bas niveau doit reposer sur des langages dédiés ,tel que *Listz*, qui utilisent des concepts métiers et, doit s'appuyer sur des plateformes comme *Arcane* pour la gestion des services annexes (jeux de données, sorties, maillages, partitionnement). Finalement, nous considérons que de plus en plus de décisions ne pourront être prises qu'au moment de l'exécution afin d'exploiter pleinement les futures machines.

Deuxième partie

Contribution

Chapitre

4

MDE4HPC : une approche modèle pour la simulation numérique

La difficulté n'est pas de comprendre les idées nouvelles,
mais d'échapper aux idées anciennes.

John Maynard Keynes.

Sommaire

Nous devons trouver une solution face aux limites des solutions actuelles à gérer les problèmes de dépendances aux architectures, de mélange des préoccupations et de complexité de programmation. L'ingénierie dirigée par les modèles (*IDM*) est reconnue comme une solution capable de gérer ce type de problèmes [188, 91]. Notre contribution s'inscrit dans ce cadre et propose l'application de l'approche *IDM* au développement d'applications de simulation numérique haute-performance.

Ce chapitre a pour objectif de présenter notre proposition, l'approche *MDE4HPC*. Dans un premier temps, nous aborderons les fondements théoriques sur lesquels elle s'appuie : l'ingénierie dirigée par les modèles. Puis, dans un second temps, nous présenterons les caractéristiques et la spécificité de notre approche.

4.1 Fondements théoriques

Depuis l'époque où Goldstine et Von Neumann utilisaient des diagrammes de *flow* [97] pour planifier leurs programmes (voir figure 4.1), en passant par les années 1970 avec le formalisme *SADT* [180] ou plus récemment avec le langage *UML*, les modèles ont toujours été présents dans les développements logiciels afin de faciliter la conception et la compréhension des applications devant être produites. Pourtant, ces modèles ont toujours fait l'objet de critiques. On peut citer, par exemple, le problème de la charge de travail nécessaire au maintien de la cohérence entre les modèles et le code source lors de la prise en compte d'évolutions. Ces critiques sont majoritairement liées au fait que ces modèles soient uniquement utilisés dans une optique de communication et de compréhension entre des acteurs humains. Favre et coll. caractérisent ces modèles de « contemplatifs » [85].

L'ingénierie dirigée par les modèles (*IDM*) propose de produire tout ou une partie d'un logiciel à partir de modèles. Les modèles deviennent ainsi des entités de premier plan dans le

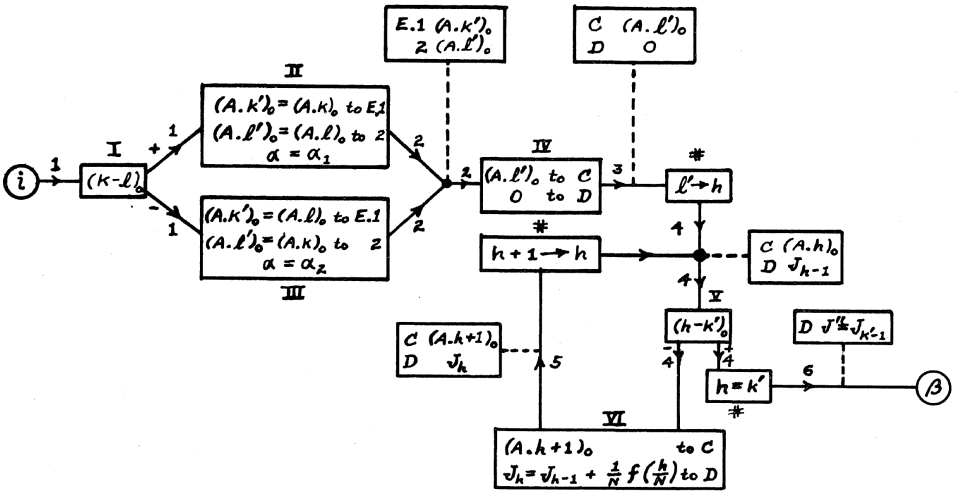


FIGURE 4.1 – Diagramme de *flow* décrivant le comportement d'une méthode d'intégration [97]

développement logiciel. En plus d'être « contemplatifs », ils sont désormais « productifs » : ils sont interprétés et manipulés via des transformations de modèles en phase de production.

Plus globalement, on peut voir l'*IDM* comme une famille d'approches développées par des laboratoires de recherches et des industriels [85]. Coté universitaire, on peut citer les travaux du laboratoire *ISIS*[1] sur le *model-integrated computing* (MIC) [199] et, coté industrie, les travaux de *Microsoft* sur les *software factories* [102]. Mais les travaux qui ont le plus influencé la communauté *IDM* sont issus de l'effort de standardisation du consortium *OMG* au travers de l'approche *Model-Driven Architecture* (*MDA*). Son succès est tel, qu'il entraine une confusion des genres, réduisant fréquemment à tort, l'*IDM* au seul *MDA*.

Avant de présenter les standards et recommandations que propose l'*OMG* dans le cadre de l'*IDM*, nous allons aborder les principes généraux et notamment clarifier un point important : la notion de modèle.

4.1.1 Principes généraux de l'*IDM*

Modèles

> *« Models have never been invented, they have been around (at least) since humans started to exist. Therefore, nobody can just define what a model is, and expect that other people will accept this definition ; endless discussions have proven that there is no consistent common understanding of models. »*
>
> J. Ludewig [141]

À l'image de cette remarque de Ludewig, il semble difficile de donner une définition universelle de la notion de modèle tellement les opinions sur la question sont nombreuses [36, 187, 138, 160]. Toutefois, dans le cadre de notre étude, nous raisonnerons à partir de la définition d'un modèle que Stachowiak établit au travers de ses caractéristiques :

1. *Institute for Software Integrated Systems* http://www.isis.vanderbilt.edu/research/MIC

> « *A model needs to possess the following three features :*
> – *Mapping feature. A model is based on an original.*
> – *Reduction feature. A model only reflects a (relevant) selection of an original's properties*
> – *Pragmatic feature. A model needs to be usable in place of an original with respect to some purpose.*
> »
>
> H. Stachowiak [194]

Nous en déduisons qu'un modèle est une représentation partielle d'un système étudié conçue pour un certain usage. On dit que le système est représenté par le modèle (relation notée « EstReprésentéPar »). Or dans le cadre de l'*IDM*, il est nécessaire que cette représentation soit manipulable par une machine. Nous complétons donc notre première définition en ajoutant une quatrième caractéristique : un modèle « compatible *IDM* » doit être une représentation décrite dans un langage bien défini. C'est dans ce cadre qu'intervient la notion de métamodèle.

Métamodèle

Un métamodèle est modèle d'un langage de modélisation [83]. Il fournit un cadre et des concepts permettant de définir des modèles bien formés vis-à-vis de ce dernier. Il existe alors une relation de conformité, nommée parfois « EstConformeA », entre le modèle et son métamodèle. À des fins de compréhension, on peut faire le parallèle entre le monde de la programmation et celui de la modélisation. On peut comparer un métamodèle à la grammaire d'un langage de programmation puisqu'on retrouve une relation de conformité entre le programme et la grammaire du langage de programmation qu'il utilise.

Transformations

Comme nous l'avons évoqué précédemment, les transformations de modèles sont le biais permettant de rendre les modèles « productifs ».

> « *A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.* »
>
> A. Kleppe et coll. [134]

Il existe plusieurs classes de transformations de modèles, chacune remplissant différents objectifs. Mens et coll. proposent une taxonomie de ces transformations [151]. Deux caractéristiques principales sont identifiées. La première concerne la relation entre le métamodèle source et destination. S'ils sont identiques alors la transformation est qualifiée d'*endogène*, dans le cas contraire, elle sera qualifiée d'*exogène*. La deuxième caractéristique concerne la variation du niveau d'abstraction. Une transformation de modèle est qualifiée d'*horizontale* si le modèle source et le modèle produit possède le même niveau d'abstraction, dans le cas contraire, elle sera qualifiée de *verticale*. La figure 4.2 montre quelques exemples de classe de transformations.

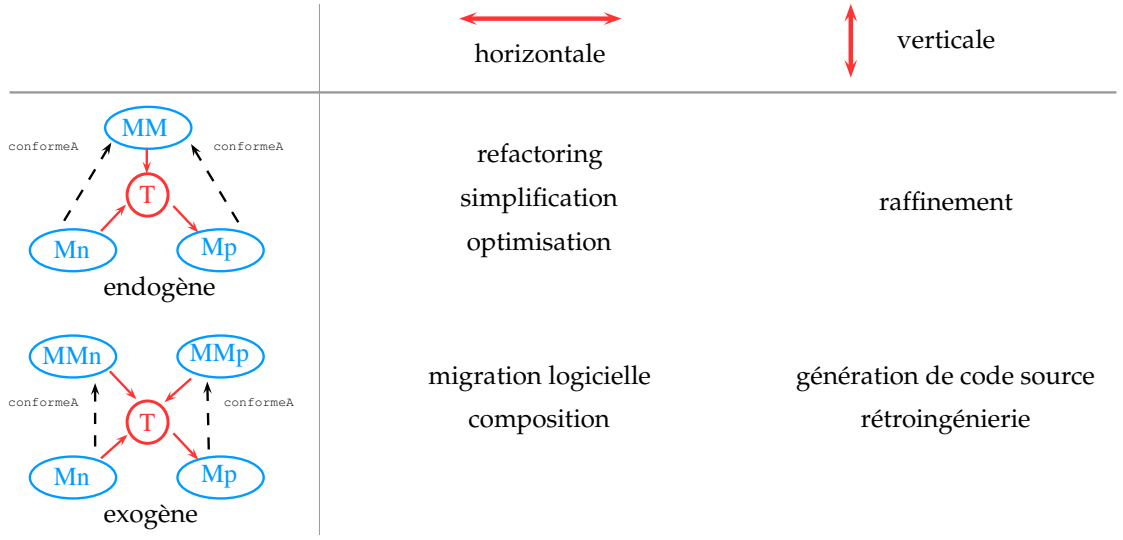


FIGURE 4.2 – Exemples de type de transformations

Langage de modélisation

Un langage de modélisation est défini par une syntaxe abstraite, une ou des syntaxes concrètes, la correspondance entre la ou les syntaxes concrètes et la syntaxe abstraite, et finalement, une sémantique [133]. La syntaxe abstraite définit la structure du langage tel qu'elle est manipulée par l'ordinateur. Une syntaxe concrète définit l'apparence du langage, c'est elle qui est manipulée par le développeur. Finalement, la sémantique définit un moyen permettant de comprendre la signification et le comportement de l'ensemble des modèles que peut produire le langage.

Les métamodèles peuvent être utilisés pour définir la syntaxe abstraite d'un langage. En ce qui concerne la syntaxe concrète et la sémantique, Kleppe dans [133] présente plusieurs méthodes plus ou moins formelles permettant de les définir. Nous n'aborderons pas ces méthodes ici.

À l'opposé du langage *UML* (section 4.1.2) qui est souvent caractérisé de généraliste, il existe une classe de langages que l'on appelle « langage dédié ». À l'instar de la mouvance des *DSL* dans les langages de programmation, les langages de modélisation dédiés, ou *Domain Specific Modeling Language* (*DSML*) en anglais, sont des langages de modélisation spécialisés pour des domaines métiers ou des problèmes particuliers. C'est le cas du langage *HPCML* présenté dans cette thèse (chapitre 5) qui a été conçu pour la description d'applications de simulation numérique haute-performance. Comme nous l'avons déjà mentionné, la syntaxe abstraite d'un *DSML* peut être spécifiée à l'aide d'un métamodèle. Cependant, il existe une autre solution au travers des mécanismes d'extension et de restriction du langage *UML* appelé profil. Cette approche n'est viable que dans le cas où le nouveau langage possède des concepts en commun avec *UML*.

4.1.2 L'approche *Model Driven Architecture* de l'*OMG*

L'*OMG*[2] est un consortium créé en 1989 et regroupant des industriels et des académiques dont l'objectif initial était la mise en place de standards pour la conception de systèmes logiciels orientés objet répartis. C'est en 1997 que l'*OMG* se tourne vers la modélisation avec

2. http://www.omg.org/

la création du standard *Unified Modeling Language* (*UML*) [110, 111]. *UML* est un langage de modélisation généraliste destiné aux applications orientées objet. Il est devenu l'un des langages de modélisation les plus utilisés.

En 2000, l'*OMG* proposa l'approche *Model-Driven Architecture* (*MDA*) [193, 39] afin de fournir des recommandations pour les développements à base de modèles. Dans cette optique, *MDA* englobe la définition de plusieurs standards tel que *MOF*, *OCL*, *QVT* et *XMI*.

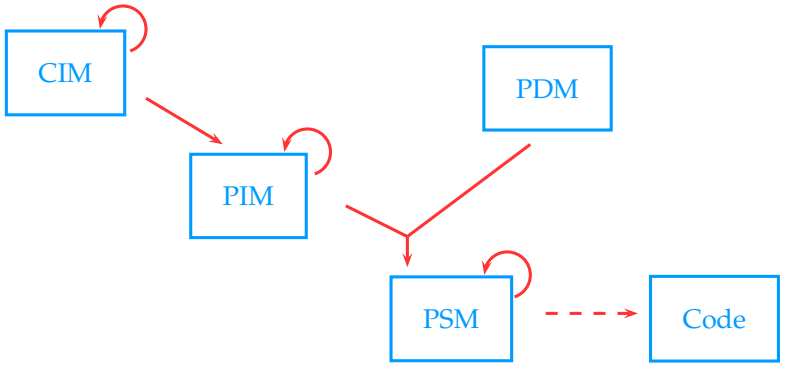


FIGURE 4.3 – Aperçu du processus global préconisé par l'approche *MDA*

En se basant sur ces standards, l'approche *MDA* préconise la définition de plusieurs modèles afin de générer le code de l'application. Parmi ces modèles présentés dans la figure 4.3, on distingue :

- les *Computation Independent Models* (*CIM*) qui sont des modèles d'exigences.
- les *Platform Independent Models* (*PIM*) qui sont des modèles d'analyse et de conception devant, en théorie, découler, au moins partiellement, des *CIM*. Ces modèles ont aussi pour vocation d'assurer la pérennité de l'application, puisqu'ils décrivent le savoir métier sans corrélation avec les plateformes d'exécution.
- les *Platform Description Models* (*PDM*) qui sont des modèles décrivant les caractéristiques des plateformes d'exécution.
- les *Platform Specific Models* (*PSM*) qui sont des modèles spécialisés pour une plateforme d'exécution particulière. Ils sont obtenus par des transformations combinant les *PIM* avec les informations des plateformes contenues dans les *PDM*. A l'opposé des *PIM*, les *PSM* sont considérés comme des modèles jetables qui ne jouent qu'un rôle d'intermédiaire pour faciliter la génération du code de l'application finale.

La suite de cette section sera consacrée à la présentation des différents standards mis en avant par l'approche *MDA*.

Meta-Object Facility (*MOF*)

Malgré son succès, il était clair qu'*UML* possédait des limites et que d'autres métamodèles adressant des besoins différents seraient nécessaires. Afin d'éviter une prolifération de métamodèles incompatibles évoluant indépendamment, il était nécessaire de mettre en place un cadre permettant de formaliser la définition des métamodèles [35]. La solution fut de proposer un langage pour définir des métamodèles, c'est-à-dire un metametamodèle, appelé *Meta-Object Facility* (*MOF*) [108].

La figure 4.4 présente les niveaux de modélisation proposés par *MDA*. On retrouve le *MOF* au sommet de la pyramide. Le *MOF* est son propre métamodèle. On dit qu'il est réflexif car il est capable de se décrire à partir de ses propres concepts pour éviter la nécessité d'une infinité de niveaux. En dessous, chaque métamodèle définit un langage par

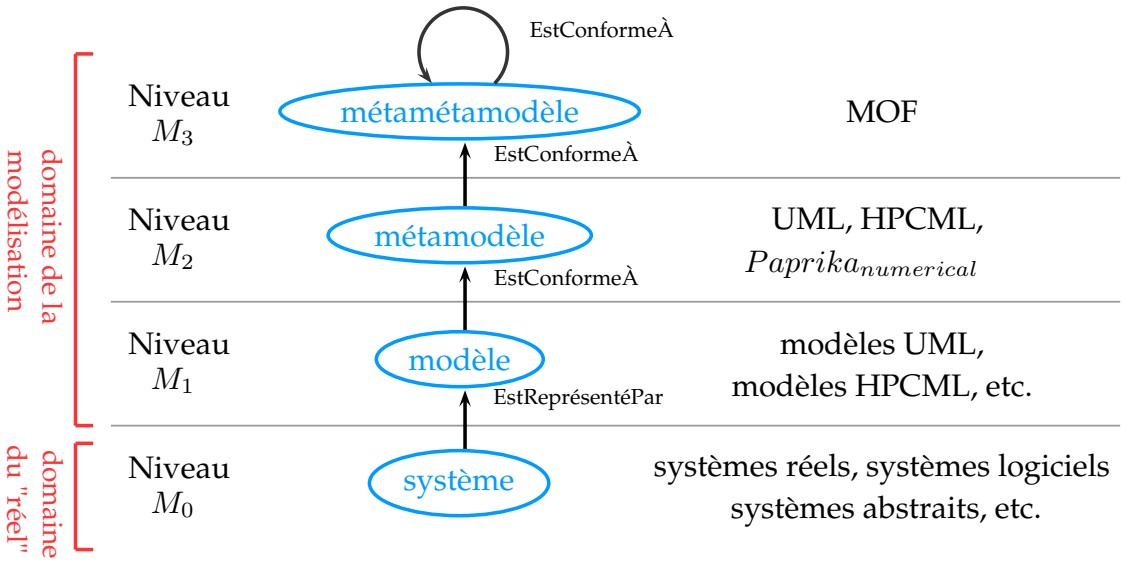


FIGURE 4.4 – Structure des différents niveaux de modélisation de l'approche *MDA*

domaine d'intérêt. On y retrouve *UML* qui comme on l'a déjà évoqué permet de modéliser les artefacts d'un système logiciel orienté objet. C'est aussi à ce niveau que se situe le langage *HPCML* que nous introduirons au chapitre 5.

Object Constraint Language (*OCL*)

Les concepts fournis par un métamodèle n'offrent pas toujours l'expressivité nécessaire pour spécifier les relations ou propriétés des modèles qu'il décrit. Au passage, nous rappelons que le *MOF* est aussi un métamodèle et que le problème se pose donc à plusieurs niveaux de modélisation. C'est en vue de combler ce manque, que l'*OMG* a proposé l'*Object Constraint Language* (*OCL*) [113]. C'est un langage fortement typé dont l'objectif est de fournir un mécanisme permettant d'exprimer des contraintes pour compléter la sémantique statique d'un métamodèle.

L'évaluation d'une expression *OCL* ne provoque aucun changement d'état du modèle sur lequel elle est évaluée. Par conséquent, le langage *OCL* ne génère aucun effet de bord. On trouve trois catégories principales d'expressions en *OCL* :

- les *invariants* qui permettent de spécifier une propriété de classe devant toujours être vraie.
- les *préconditions* qui permettent de spécifier les conditions d'appel d'une méthode au travers d'une propriété devant être vérifiée avant l'appel de cette méthode.
- les *postconditions* qui permettent de spécifier le résultat attendu par une méthode au travers d'une propriété devant être vérifiée après l'appel de cette méthode.

Query/View/Transformation (*QVT*)

Comme les transformations de modèles sont un des piliers de l'*IDM*, l'approche *MDA* se devait de proposer une façon standardisée pour les spécifier. Le standard *Query/View/Transformation* (*QVT*) [109] joue ce rôle. Il définit trois langages permettant d'exprimer des transformations de modèles à modèles. On trouve un langage de type impératif appelé *QVT-Operational*, deux langages déclaratifs nommés *QVT-Core* et *QVT-Relation* qui se

trouvent à des niveaux d'abstractions différents. En addition, il existe un mécanisme d'extension appelé *QVT-BlackBox*, qui permet de décrire des fonctionnalités de transformations dans un langage externe.

XML Metadata Interchange (*XMI*)

Les modèles sont des entités abstraites qui, par conséquent, possèdent une infinité de représentations informatiques possibles. Par soucis d'interopérabilité et de pérennité, il était nécessaire d'utiliser un format unique et standardisé. C'est pour ces raisons que l'*OMG* a proposé le standard *XML Metadata Interchange* (*XMI*) [112]. Il a pour objectif de définir un moyen permettant de stocker un modèle sous forme de document *XML*.

4.2 Définition de l'approche *MDE4HPC*

L'approche *MDE4HPC* est notre proposition concernant la façon d'appliquer les principes généraux de l'*IDM* et de ceux du *MDA* au développement d'applications de simulation numérique haute-performance.

Après une présentation du cadre général dans lequel intervient l'approche *MDE4HPC*, nous introduirons ses concepts et ses recommandations.

4.2.1 Analyse de la situation

Afin de comprendre l'organisation du développement proposée par l'approche *MDE4HPC*, il est tout d'abord nécessaire d'éclaircir deux points. Comment sont utilisés les codes de simulation ? Comment sont-ils développés ?

Réalisation d'un simulation numérique

La figure 4.5 introduit quelques éléments permettant de répondre à la première question. Elle décrit le processus de conduite d'une simulation numérique réalisée par un physicien en se basant sur le formalisme *SPEM* [5]. Nous avons modélisé ce processus à partir d'une analyse des pratiques au sein du *CEA/DAM*. On remarque que l'on peut diviser le processus en trois grandes phases :

- *phase pré-calcul*. Dans cette phase, le physicien prépare les données nécessaires à la simulation. On identifie traditionnellement deux catégories parmi ces données. D'un coté, il y a les données initiales du problème. Le maillage, qui décrit la discrétisation spatiale du système étudié, entre dans cette catégorie. De l'autre coté, on distingue les données permettant de paramétrer le modèle physique. Notons toutefois qu'il existe souvent des paramètres additionnels permettant de contrôler le modèle de résolution mathématiques mais ils sont cachés pour des calculs réalisés en production. L'ensemble de ces données est appelé « jeu de données ».
  Le physicien doit aussi décider, au travers du logiciel de soumission de cas de calcul, dans quelles conditions le code sera exécuté. Ces conditions comprennent le type et la quantité des ressources d'exécution nécessaires pour le calcul. Par exemple, l'utilisateur peut demander la réservation de 64 cœurs sur un nœud de calcul à mémoire partagée. Un cas de calcul référence un ou plusieurs jeux de données.
- *phase de calcul*. Le calculateur exécute le code de simulation en fonction des paramètres fournis par l'utilisateur. Cette phase peut durer de quelques minutes à plusieurs jours.

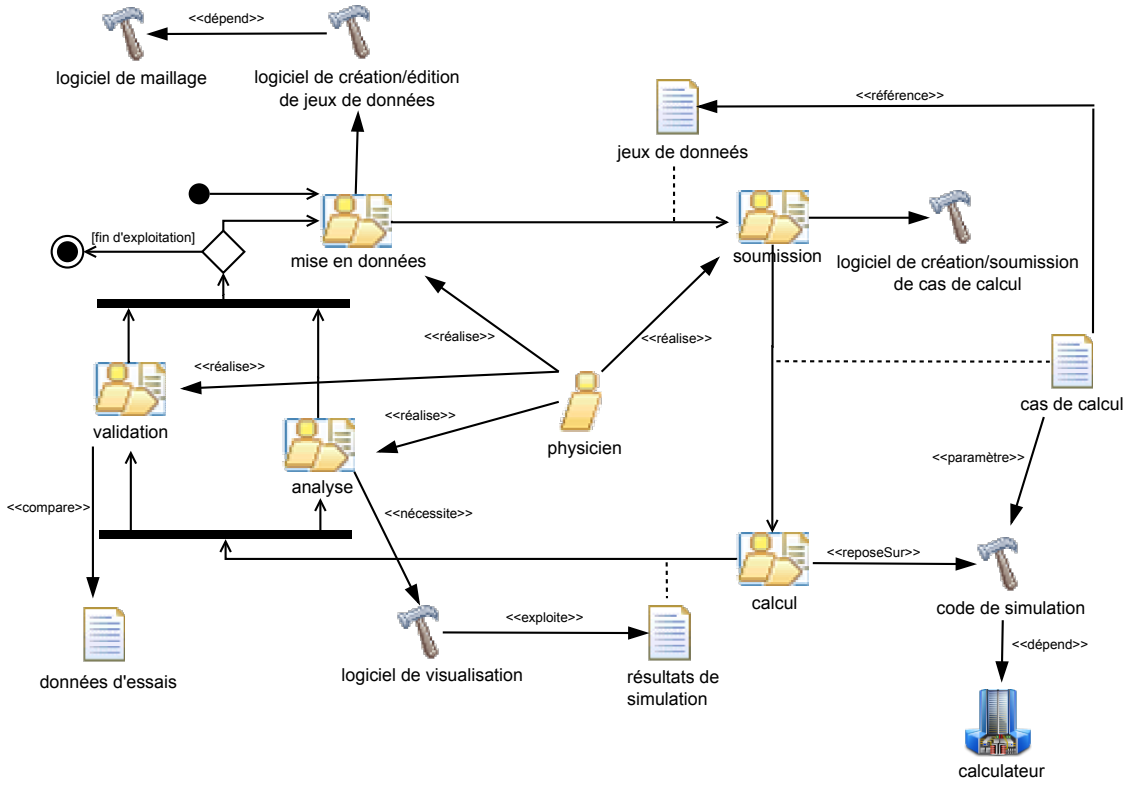


FIGURE 4.5 – Processus de conduite d'une simulation numérique par un physicien

– *phase post-calcul*. Cette dernière phase concerne la récupération et l'exploitation des résultats. Deux types d'activités sont regroupées dans cette phase. Il y a tout d'abord l'analyse des résultats à proprement parler. Elle permet de répondre aux questions que l'on se pose sur le système étudié. Il y a ensuite l'activité de validation qui consiste à vérifier les résultats produits et à minimiser l'erreur des modèles physiques et mathématiques par rapport à la « réalité ». Pour ce faire, on va recaler les modèles en utilisant des données provenant d'expérimentations.

De cette analyse du processus d'une simulation numérique découle un premier constat. Un code de simulation numérique n'est pas un élément logiciel isolé. Il existe tout un panel d'artefacts logiciels dont il dépend ou qui dépende de lui. L'ensemble de ces relations va donc impacter les étapes de développement et de maintenance du code de simulation.

Développement d'un code de simulation numérique

Nous avons déjà abordé le sujet des cycles de développement rencontrés dans le cas des codes de simulation dans l'état de l'art (section 3.2.2). Nous nous intéressons ici plus particulièrement à l'identification des intervenants et de leurs contributions. On identifie, du moins dans le cas du *CEA/DAM*, trois métiers différents intervenant dans le développement d'un code de simulation :

– le *physicien* qui a pour rôle de définir le besoin au travers du modèle physique (figure 2.1).
– le *numéricien* qui va définir et implémenter un modèle mathématique de résolution du modèle physique (figure 2.1). C'est le plus souvent le numéricien qui aura la charge de paralléliser son implémentation.
– l'*ingénieur logiciel* qui va intervenir à plusieurs niveaux. Il va fournir des briques lo-

gicielles de base aux numériciens afin de leur faciliter la tâche. Il est aussi responsable de fournir l'écosystème logiciel gravitant autour du code de simulation tel que nous l'avons identifié dans la section précédente.

À noter qu'en fonction des projets et notamment du nombre de participants dans un projet, la frontière entre les métiers peut être plus ou moins claire. En effet, il n'est pas rare de voir des projets qui reposent sur les épaules d'« hommes orchestres » qui ont la charge de tous les aspects du développement.

Bilan des considérations à prendre en compte

Nous pouvons à présent faire un bilan des points importants que l'approche *MDE4HPC* se devra de traiter pour être adaptée à un développement de logiciel de simulation numérique.

Il faudra tout d'abord gérer les interactions entre les différents métiers (physicien, numéricien, ingénieur logiciel) et surtout s'arranger pour qu'un métier donné n'ait pas à se préoccuper des aspects d'un autre métier comme c'est le cas actuellement. On peut citer l'exemple du numéricien qui doit prendre en compte de façon précise les particularités matérielles d'un calculateur. À noter, que cette séparation des métiers n'implique pas un cloisonnement des ces derniers.

Il faudra finalement que l'approche *MDE4HPC* facilite la mise en place d'un code de simulation au sein de son écosystème logiciel.

4.2.2 Proposition d'architecture

Le principe de l'approche *MDE4HPC* est simple : modéliser les artefacts de chaque métier identifié (modèle physique, modèle mathématique numérique et aspects informatiques) et les combiner afin de générer l'application de simulation numérique. Pour cela, l'approche *MDE4HPC* propose une architecture en couches reposant sur plusieurs niveaux d'abstraction où chaque niveau d'abstraction cible un métier particulier. Les différentes couches sont présentées de façon simplifiée dans la figure 4.6.

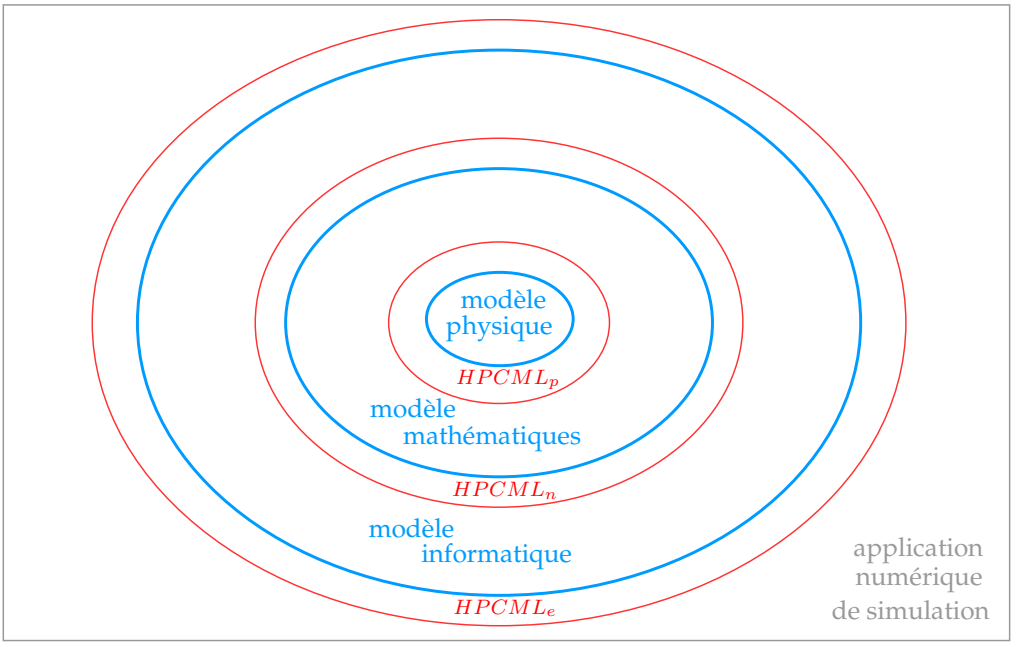


FIGURE 4.6 – Couches d'abstraction proposées par l'approche *MDE4HPC*

On note deux types d'éléments sur ce schéma, les modèles en bleu et le *High-Performance Computing Modeling Language* (*HPCML*) en rouge. On remarque aussi que l'on retrouve un type de modèle par type de métier. Le langage *HPCML* joue en fait le rôle d'intermédiaire entre les métiers. C'est pourquoi il est divisé en trois sous-langages que sont $HPCML_p$, $HPCML_n$ et $HPCML_e$.

Ces trois langages offrent une possibilité unique de vérifier la cohérence entre les trois types de modèles que nous avons identifiés. Ils permettent aussi d'envisager une traçabilité entre les concepts situés à différents niveaux d'abstraction. Nous allons maintenant détailler les caractéristiques de chacun de ces langages.

Couche dédiée au métier de physicien

Nous avons identifié le physicien comme étant le responsable de l'expression du besoin au travers de la définition du modèle physique. Le langage $HPCML_p$, où l'indice p signifie *physics*, fournit des concepts permettant de formaliser cette expression des besoins. La figure 4.7 montre la structure du langage $HPCML_p$, deux aspects y étant abordés :

- *besoin* : le physicien doit pouvoir décrire son modèle physique (équations, hypothèses, propriétés) ainsi que le cheminement qui a abouti à ce modèle.
- *validation* : le physicien doit pouvoir spécifier le domaine de validation de son modèle et indiquer sur ce domaine de validité les propriétés des résultats attendus.

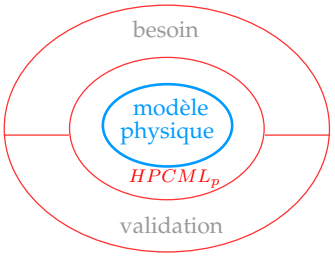


FIGURE 4.7 – Aspects du sous-langage d'*HPCML* permettant de spécifier le modèle physique

Couche dédiée au métier de numéricien

Pour le métier de numéricien, les choses se compliquent. Il se trouve en interaction avec deux autres métiers : le physicien d'un coté et l'ingénieur logiciel de l'autre. L'objectif du langage $HPCML_n$, où l'indice n signifie *numeric*, est de fournir au numéricien des concepts lui permettant de spécifier finement le modèle de résolution mathématique aussi appelé modèle numérique. Ce modèle du modèle mathématique devra découler du modèle du modèle physique.

Bien que présents, les concepts de nature informatique devront abstraire au maximum les préoccupations logicielles ou matérielles en rapport avec les plateformes d'exécution envisagées. Ce point est particulièrement important pour deux raisons. Tout d'abord, il permet d'améliorer l'accessibilité des développements aux numériciens. Ensuite, il augmente le potentiel de portabilité des applications développées avec l'approche *MDE4HPC*. Autrement dit, les modèles de ces applications s'accommoderont plus facilement de ruptures technologiques concernant les architectures matérielles et logicielles. L'identification des concepts remplissant ces conditions est une tâche ardue. En effet, il faut d'un coté monter le plus possible en abstraction et de l'autre, s'assurer que le concept est assez concret pour être capable de le spécialiser vers une plateforme d'exécution particulière.

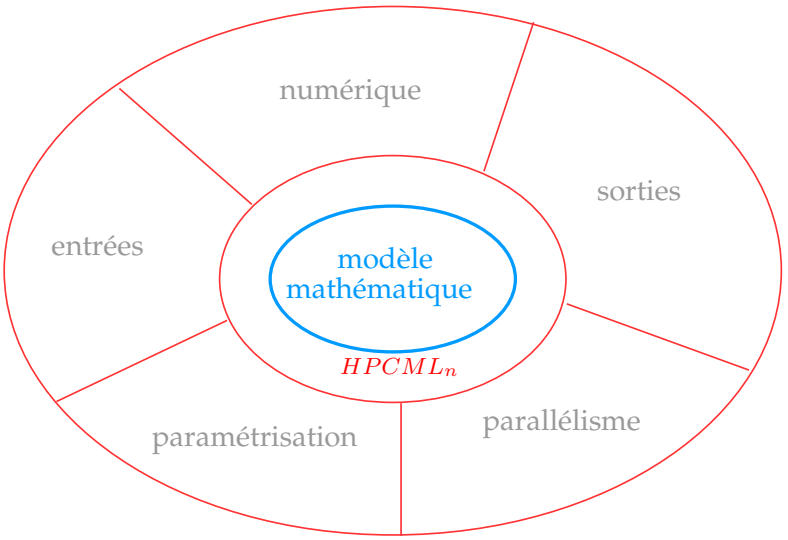


FIGURE 4.8 – Aspects du sous-langage d'*HPCML* permettant de spécifier le modèle mathématique

La figure 4.8 présente la structure du langage $HPCML_m$. Comme on l'a déjà évoqué, il permet de modéliser le modèle mathématique. Pour ce faire, il fournit des concepts permettant de traiter les différents aspects de cette tâche :

- *entrées* : le numéricien doit pouvoir spécifier de manière structurée les données servant à paramétrer son code de simulation. Ces paramètres peuvent adresser des préoccupations d'ordre physiques ou mathématiques. Le logiciel de création et d'édition de jeux de données (figure 4.5) sera directement basé sur cette partie du modèle numérique.
- *numérique* : le numéricien doit pouvoir modéliser son schéma numérique à l'aide d'un formalisme proche du langage mathématique, donc proche de son langage métier. Cette partie du langage devra faciliter la réutilisation entre les applications en fournissant des mécanismes permettant de facilement composer une application à partir de briques de base.
- *parallélisme* : le numéricien doit pouvoir indiquer les sources potentielles de parallélisme dans son application sans toutefois devoir maitriser des concepts proches d'un calculateur particulier. Par le terme parallélisme on désigne l'ensemble des mécanismes permettant d'effectuer des calculs de manière concurrente.
- *sorties* : le numéricien doit pouvoir spécifier les résultats de calcul qu'il souhaite exporter de son application. Le choix de concepts abstraits pour cette tâche doit faciliter l'intégration avec les logiciels utilisés en phase d'analyse tel que les logiciels de visualisation (figure 4.5).
- *paramétrisation* : finalement, le numéricien doit pouvoir spécifier la façon dont son application pourra être utilisée pour conduire des études paramétriques. Cette partie du modèle pourra servir à la fois pour la génération d'interfaces graphiques de conduite d'études paramétriques et pour exploiter le parallélisme que les études paramétriques offrent.

Couche dédiée au métier d'ingénieur logiciel

La tâche à accomplir par l'ingénieur logiciel est plus importante dans cette approche que lors d'un développement « traditionnel » de codes de calcul. L'approche *MDE4HPC* propose,

en effet, une séparation des concepts permettant aux différents métiers de se concentrer sur leur domaine. Une grande partie des aspects informatiques traditionnellement pris en charge par le numéricien est donc, dans le cas notre approche, sous la tutelle de l'ingénieur logiciel.

L'objectif du langage *HPCML*$_e$, où l'indice *e* signifie *execution*, est de fournir à l'ingénieur logiciel des concepts lui permettant de définir comment les modèles de résolution mathématiques doivent être implémentés dans une plateforme d'exécution cible. Dans cette tâche, le langage *HPCML*$_e$ devra être complété par des langages de transformations de modèles traditionnels.

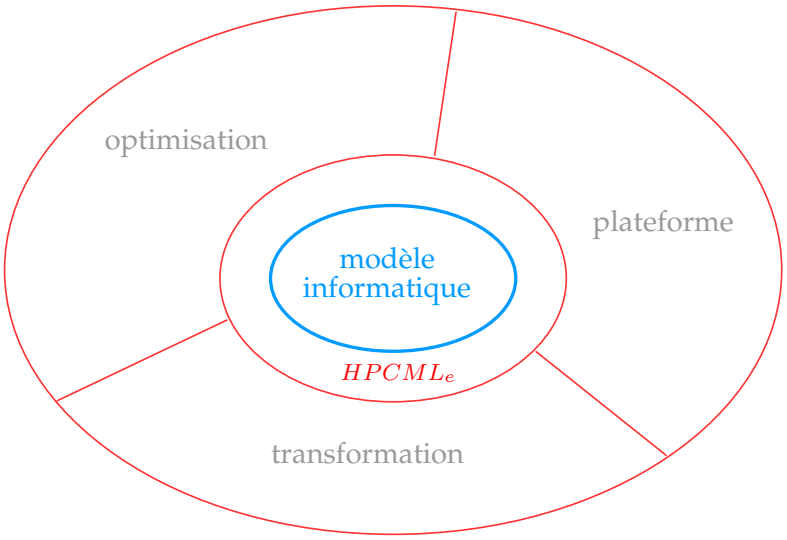


FIGURE 4.9 – Aspects du sous-langage d'*HPCML* permettant de construire le modèle informatique

La figure 4.9 présente la structure du langage *HPCML*$_e$. Afin de faciliter le travail de l'ingénieur logiciel dans la tâche de production de l'application, le langage *HPCML*$_e$ propose des concepts adressant plusieurs aspects de cette tâche.

- *optimisation* : l'ingénieur logiciel doit pouvoir indiquer sur un modèle de modèle mathématique où et quel type d'optimisation il souhaite appliquer lors de la traduction vers une machine cible.
- *plateforme* : l'ingénieur logiciel doit pouvoir décrire les caractéristiques de la plateforme d'exécution. Cette description servira à paramétrer les transformations de modèles responsables de l'implémentation du modèle mathématique.
- *transformation* : en addition des transformations génériques à tous les projets, l'ingénieur logiciel doit pouvoir décrire des transformations spécifiques à un projet particulier.

4.2.3 Processus de développement

Le processus proposé par l'approche *MDE4HPC* est présenté dans la figure 4.10. On y distingue des modèles (en bleu) basés sur les trois sous-langages que nous venons de présenter.

Le processus est globalement assez souple, puisqu'il permet de mettre à jour un modèle indépendamment de son niveau d'abstraction. De plus, il est possible de rester sur le niveau d'abstraction correspondant à son métier sans avoir besoin de lancer à chaque fois

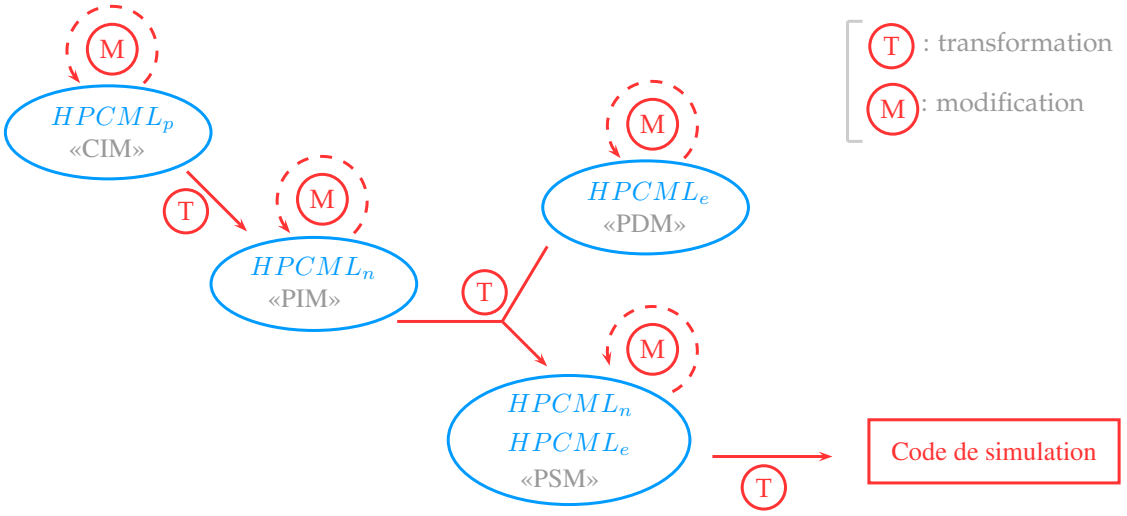


FIGURE 4.10 – Processus de développement proposé par l'approche *MDE4HPC*

les transformations produisant l'application. Ceci est rendu possible par les vérifications statiques avancées qui sont réalisables sur les modèles. Ce processus est aussi compatible pour être utilisé de façon itérative ou semi-itérative. Cette caractéristique est importante puisque nous avions identifié, dans la section 3.2.2, ce type de processus comme étant le plus adapté au développement de codes de simulation.

Notons que ce processus de développement s'adapte particulièrement bien aux différents types de maintenance. Par exemple, lors d'un changement de calculateur (maintenance adaptative), seules les étapes de plus bas niveaux seront impactées (modèles et transformations dépendants du langage $HPCML_e$).

Sur la figure 4.10, on remarque aussi que le processus *MDE4HPC* peut être facilement aligné sur les recommandations de l'approche *MDA* (correspondances approximatives décrites entre chevrons sur la figure).

4.3 Conclusion

Dans ce chapitre nous avons abordé les principes fondamentaux de l'approche *MDE4HPC* qui s'inspire de l'ingénierie dirigée par les modèles (*IDM*). *MDE4HPC* propose, d'une part, de modéliser les applications de simulation numérique et, d'autre part, de générer par transformations les implémentations logicielles adaptés aux architectures cibles. On retiendra notamment le principe des trois couches d'abstraction qui visent chacune un métier particulier.

La définition et la validation des trois sous-langages d'*HPCML* représentent des tâches dont l'ampleur sort du cadre d'une simple thèse. Par conséquent, et afin de néanmoins valider l'approche *MDE4HPC* partiellement, nous avons fait les choix suivants. Nous nous sommes tout d'abord concentrés sur la définition du langage $HPCML_n$ car il se trouve au carrefour des deux métiers et qu'il concentre le plus de verrous technologiques, notamment l'abstraction du parallélisme. Afin de vérifier sa pertinence, nous avons développé un générateur capable de traduire ces modèles en application de simulation numérique exécutable. Ce générateur repose sur des transformations de modèles que l'on pourrait caractériser de « monolithiques » et ne sont pas paramétrables finement via un *PDM* comme proposé par *MDE4HPC*.

Par conséquent, dans le chapitre suivant nous présenterons le langage *HPCML*$_n$. Dans le reste de la partie « Contribution », l'emploi du terme *HPCML* fera souvent référence à la seule couche *HPCML*$_n$. Puisque les recherches effectuées sur les langages *HPCML*$_p$ et *HPCML*$_e$ sont incomplètes, nous les présenterons dans le chapitre 9 de la partie « Perspectives ».

Chapitre

5

Langage HPCML

L'architecture, c'est formuler les problèmes avec clarté.
Le Corbusier.

Sommaire

Selon Bézivin et coll. « *A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system* » [36]. L'objectif du métamodèle d'$HPCML_n$ est de fournir aux numériciens la possibilité de spécifier de façon claire et pérenne leur modèle de résolution mathématique.

Ce chapitre présente le langage *high-performance computing modeling language* (*HPCML*) qui se trouve au cœur de la phase de conception de l'approche *MDE4HPC*. La présentation du langage se décompose en deux parties abordant respectivement sa syntaxe abstraite et sa syntaxe concrète.

5.1 Syntaxe abstraite

Cette section a pour objectif de présenter la syntaxe abstraite du langage *HPCML* qui a été formalisée par un métamodèle. Dans un premier temps, nous présenterons brièvement la démarche suivie pour l'élaboration de ce métamodèle ainsi que sa structure générale. Dans un second temps, nous introduirons le contenu des différents paquetages composant ce métamodèle.

5.1.1 Présentation générale

Le métamodèle d'*HPCML* présenté dans ce chapitre est le résultat d'un processus itératif qui s'est nourri d'informations provenant de sources diverses : codes de simulation existants,

bibliothèques métier, langages existants, documentations techniques, discussions avec des experts métier. Alors qu'une partie du métamodèle découle de cette analyse de l'existant, une autre partie s'en détache afin de proposer une approche nouvelle pour répondre à notre problématique. Globalement, on peut dire que le métamodèle propose une approche novatrice tout en essayant de conserver des concepts proches des habitudes des numériciens et de leurs outils.

Lors des premières réflexions sur le langage *HPCML*, nous avons rapidement pris conscience qu'il serait assez éloigné du langage *UML*. De plus, nous souhaitions obtenir un langage avec le minimum de concepts. À partir de ce constat, nous avons conclu que l'utilisation d'un profil *UML* n'était pas pertinente dans notre cas de figure. Nous avons donc opté pour la définition du langage par le biais d'un métamodèle. Ce dernier nous a, de surcroit, offert la possibilité de définir une sémantique de façon libre. Certaines petites parties du métamodèle sont parfois inspirées de celui d'*UML*. Ce choix s'inscrit aussi dans la tendance de fond que l'on trouve dans l'industrie sur l'utilisation des *DSML* [123].

Le métamodèle d'*HPCML* a été défini grâce au métamétamodèle *ECore* [195]. Nous aborderons les motivations de ce choix dans la section 6.1 du chapitre consacré à l'outil *ArchiMDE*. L'ensemble des diagrammes que nous présenterons dans la suite de ce chapitre respectera donc ce formalisme.

À son plus haut niveau, le métamodèle se décompose en six paquetages :

- *kernel*. Ce paquetage rassemble l'ensemble des concepts de base utilisés par les autres paquetages. Il contient notamment les concepts en rapport avec la définition des types, des maillages et de la documentation (section 5.1.2).
- *structure*. Ce paquetage fournit des concepts permettant de construire des composants. C'est l'aspect structurel des applications qui est ici géré (section 5.1.3).
- *behavior*. Ce paquetage concerne l'aspect dynamique des applications. Il contient des concepts permettant d'orchestrer les composants définis à l'aide des concepts du paquetage *structure* (section 5.1.4).
- *output*. Ce paquetage traite la définition des sorties d'un code de calcul (section 5.1.5).
- *validation*. Ce paquetage rassemble des concepts permettant de faciliter la validation des applications produites (section 5.1.6).
- *parametric*. Ce paquetage permet, au travers de ses concepts, de construire des études paramétriques (section 5.1.7).

Nous allons maintenant découvrir en détails la composition de chacun de ces six paquetages. Cette présentation s'accompagne de trois types de remarques :

- des règles de modélisation (icône) qui sont des recommandations permettant de guider et d'uniformiser le processus de modélisation. Certaines de ces règles peuvent être exprimées en langage *OCL* et faire l'objet de vérifications au sein d'un outil implémentant le langage ;
- des règles de *refactoring* (icône) qui identifient des modifications courantes ainsi que leurs impacts sur le reste des modèles ;
- des listes d'éléments de modèles par défaut (icône). Ces éléments de modèles conforment au métamodèle d'*HPCML* doivent être fournis avec les implémentations du langage *HPCML*.

5.1.2 Paquetage *kernel*

Le contenu du paquetage *kernel* est présenté dans la figure 5.1. On découvre sur cette dernière l'un des concepts les plus basiques du métamodèle : `HPCNamedElement`. Cette classe abstraite, dont de nombreux concepts héritent, définit le concept d'élément nommé.

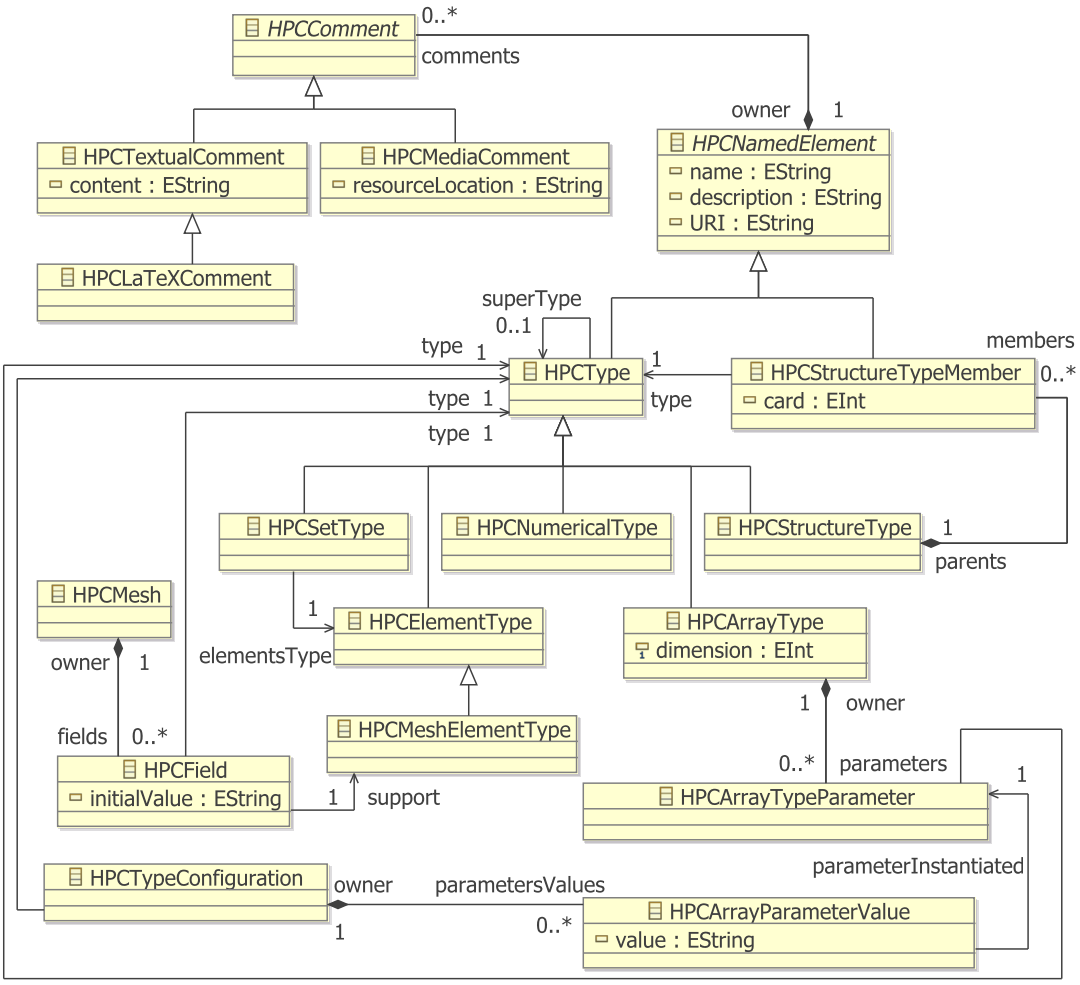


FIGURE 5.1 – Paquetage *kernel* du métamodèle d'*HPCML*

Commentaires

Il est possible de définir des commentaires (`HPCComment`) sur un `HPCNamedElement`. Ces commentaires doivent permettre de documenter le fonctionnement du code modélisé ainsi que d'expliquer ou de justifier des choix de modélisation mathématique. C'est pour cette raison qu'il est possible de créer des commentaires de diverses natures : du texte pour les remarques les plus basiques (`HPCTextualComment`), des formules mathématiques en utilisant la syntaxe du langage LaTeX[139] (`HPCLaTeXComment`) ou des médias plus riches comme des graphiques, des vidéos ou des références vers des documents entiers (`HPCMediaComment`).

Types de données

Les types de données sont un des éléments clés d'un code de calcul. Comme il n'existe pas, le plus souvent, de types complexes standards dans les langages parallèles, les développeurs créent leurs propres structures de données afin de répondre aux exigences du problème modélisé.

Cette approche possède deux inconvénients liés à la compréhension de ces structures de données. Tout d'abord, lors d'opérations de maintenance, corrective comme évolutive, il est

souvent nécessaire de comprendre ces structures de données afin de saisir le fonctionnement global du code. Ce point peut notamment poser problème en cas de rotation des équipes de développement. Ensuite l'utilisation de structures de données personnelles limite les possibilités d'optimisation car leur fonctionnement est étranger au compilateur.

On peut envisager d'autres possibilités permises par l'utilisation de types complexes standards, comme, par exemple, l'automatisation de la vérification de propriétés spécifiques aux types. Prenons le cas d'une matrice devant être définie positive. On peut imaginer une version de l'application dédiée à la « validation » et qui vérifierait, entre autres, la validité de cette propriété à plusieurs moments de l'exécution du programme.

Il était donc primordial qu'*HPCML* fournisse des mécanismes permettant de définir des types de données abstraits communs à tous les codes de simulation. Nous allons maintenant décrire les concepts choisis pour répondre à ce besoin.

Types de base

La racine de la hiérarchie des classes de types est `HPCType`. Il permet de définir les types les plus basiques. Par défaut, deux éléments sont fournis.

Bibliothèque de modèles - `HPCType`

String, Boolean

Il est possible de définir une relation d'héritage entre deux types grâce à la relation `superType`. Cette relation définit qu'un type B héritant d'un type A équivaut à dire que le type B est un sous-type du type A et qu'il hérite des attributs de A. En regardant le métamodèle du paquetage *kernel* dans la figure 5.1, on constate que la possibilité de définir des attributs sur les types est rare, seuls les concepts `HPCStructureType` et `HPCArrayType` possèdent un tel mécanisme. L'objectif de cette relation est principalement de pouvoir décrire une hiérarchie dans les types afin de permettre l'exploitation du polymorphisme.

Types numériques simples

Les types numériques sont sans nul doute ceux possédant le rôle plus important dans un code de simulation puisque tous les calculs modifient principalement l'état de variables possédant un type numérique. Le concept parent pour les types numériques est `HPCNumericalType`. Les trois types numériques simples standards sont fournis par défaut.

Bibliothèque de modèles - `HPCNumericalType`

Integer, Real, Complex

Afin de diminuer les erreurs provenant de l'assemblage de fonctions incompatibles, nous souhaitions introduire un typage fort dans *HPCML*. Ce typage fort se traduit notamment par l'utilisation de sous-types du type *Real* représentant les grandeurs physiques des variables manipulées. Imaginons que l'on définisse une interface de fonction ayant pour paramètre l'angle d'incidence d'un faisceau lumineux, ce paramètre sera défini de préférence avec le type *PlaneAngle* plutôt qu'avec son type parent *Real*.

À la différence de langages comme *SysML* avec ses *Value Type* [114], nous avons choisi de ne pas offrir la possibilité de spécifier l'unité d'un type. En effet, les équations physiques sont indépendantes des unités et ne raisonnent qu'en termes de dimensions. Une méthode

correctement définie, c'est-à-dire sans constante interne non déclarée comme telle, sera compatible avec plusieurs unités du moment qu'il existe une cohérence d'unités entre les différents paramètres. Par exemple, toutes les longueurs sont exprimées en mètre. Les unités des résultats d'une fonction sont donc dépendantes des unités des paramètres d'entrées.

C'est pourquoi la bibliothèque de modèles par défaut propose une collection de sous-types de *Real*. Ces éléments sont sujets à évolution en fonction des besoins des projets.

Bibliothèque de modèles - `Real`

Length, Mass, Time, ElectricCurrent, ThermodynamicTemperature, SubstanceAmount, Area, Volume, Speed, Acceleration, WaveNumber, MassDensity, SpecificVolume, CurrentDensity, MagneticFieldStrength, Luminance, PlaneAngle, SolidAngle, Frequency, Force, Pessure, Energy, Power, ElectricCharge, ElectricResistance, ElectricConductance, MagneticFluxDensity

Règle de modélisation

Il est conseillé d'utiliser, lors du choix du type d'une variable ou d'un paramètre, le type correspondant à sa grandeur physique plutôt que d'utiliser le type parent *Real*. Cette rigueur dans la modélisation permet d'améliorer la qualité des vérifications statiques effectuées lors de la définition d'un `HPCFlowDescriptor`.

Notons qu'il n'existe aucun concept relatif à la précision des types de données numériques. En effet, ces considérations sont purement informatiques et varient en fonction des plateformes. L'association d'un type de données abstrait à un type de donnée machine doit donc être réalisée lors du processus de transformation.

Types composés

Le concept `HPCStructureType` permet de définir des structures de types composés qu'il est possible de construire en combinant des types existants via le concept `HPCStructureTypeMember`.

La bibliothèque de modèles par défaut fournit quatre types composant des types numériques. Il est néanmoins possible de définir des types composés à partir d'autres types que les types numériques.

Bibliothèque de modèles - `HPCStructureType`

- *Real2* (x :*Real*, y :*Real*) : coordonnée 2D ;
- *Real3* (x :*Real*, y :*Real*, z :*Real*) : coordonnée 3D ;
- *Real2x2* (x :*Real2*, y :*Real2*) : tenseur 2D ;
- *Real3x3* (x :*Real3*, y :*Real3*, z :*Real3*) : tenseur 3D.

Types à base de tableaux

Le concept `HPCArrayType` permet de définir des types de tableaux paramétrables (`HPCArrayTypeParameter`) et de dimensions variables. L'instanciation des paramètres de ces types est réalisé grâce au concept `HPCTypeConfiguration` qui référence un type et stocke, le cas échéant, la valeur de ces paramètres (`HPCParamaterValue`).

Bibliothèque de modèles - `HPCArrayType`

– *Vector* (size :*Integer*) : vecteur générique ;
– *Matrix* (ncol :*Integer*, nrow :*Integer*) : matrice générique.

Il est évidemment possible de profiter du mécanisme d'héritage entre types pour définir des sous-classes de *Matrix*. Citons, par exemple, le cas du type « matrice symétrique ».

Maillage

Les équations aux dérivées partielles *EDP* sont fortement présentes dans plusieurs domaines de la physique. On les retrouve par exemple en électromagnétisme ou en mécanique des fluides. La résolution de ces équations par la voie analytique n'est que rarement possible. On fait alors appel à une résolution numérique. Il existe plusieurs modèles mathématiques de résolution de ce type d'équations parmi lesquels on trouve la méthode des volumes finis, la méthode des différences finies ou encore celle des éléments finis. Ces méthodes permettent de résoudre de manière approchée les *EDP* en discrétisant le milieu du problème avec un maillage. La figure 5.2 montre, l'exemple d'une pièce en 3D partiellement maillée à l'aide de tétraèdres.

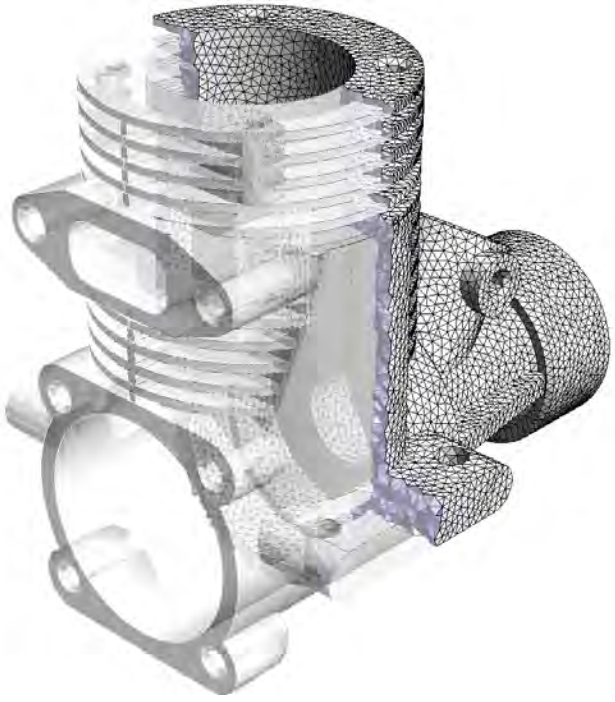


FIGURE 5.2 – Exemple de maillage 3D tétraédrique [1]

Dans ces modèles de résolution mathématique, les éléments de maillage sont des entités abstraites incontournables. Le concept `HPCMeshElementType` permet de définir les différents types d'éléments de maillage. La bibliothèque de modèles en compte quatre.

1. source : http://geuz.org/gmsh/, EDF R&D

Bibliothèque de modèles - `HPCMeshElementType`

- *Node* : sommet, élément de dimension 0 ;
- *Edge* : arête, élément de dimension 1 ;
- *Face* : face, élément de surface séparant l'espace en deux dans le cas d'un maillage 3D ;
- *Cell* : maille, élément de dimension 2 dans le cas d'une maillage 2D ou élément de dimension 3 dans le cas d'une maillage 3D.

Ces quatre types d'éléments de maillage sont représentés dans la figure 5.3.

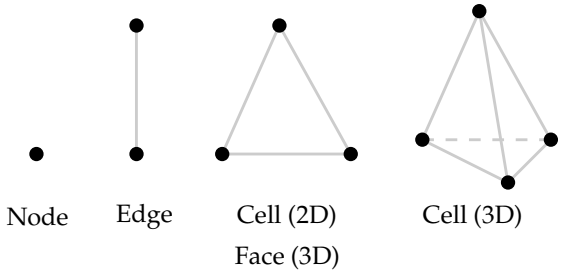


FIGURE 5.3 – Types d'éléments de maillage

Il est fréquent de manipuler des groupes d'éléments de maillage plutôt que le maillage dans son intégralité. Le concept `HPCSetType` permet, entre autres, de définir ces types d'ensemble. Par défaut, on retrouve les quatre types d'ensemble qui se composent des types d'éléments de maillage présentés précédemment.

Bibliothèque de modèles - `HPCMeshElementType`

- *NodeSet* : ensemble de sommets ;
- *EdgeSet* : ensemble d'arêtes ;
- *FaceSet* : ensemble de faces ;
- *CellSet* : ensemble de mailles, un maillage bien que global n'est en fait qu'un *CellSet* particulier.

Les modèles mathématiques abordés ici nécessitent souvent la définition de valeurs qui sont associées à des types d'éléments de maillage. Par exemple, on peut décider d'associer une vitesse (variable de type *Speed*) à chaque sommet du maillage. Les concepts `HPCMesh` et `HPCField` permettent de définir une telle structure.

5.1.3 Paquetage *structure*

Un code de simulation numérique est fonctionnel par nature. Le langage *Fortran*, qui reste la référence dans la communauté du calcul scientifique, est le reflet de cette nature fonctionnelle et décompose un programme en une hiérarchie de fonctions. Le paquetage *structure* tente à son tour de s'inspirer de cette nature fonctionnelle tout en essayant de favoriser la réutilisation entre projets et la compréhension de projets complexes. Pour cela, il propose des concepts permettant de définir et d'organiser des composants logiciels dédiés à la simulation numérique. Son contenu est présenté dans la figure 5.4.

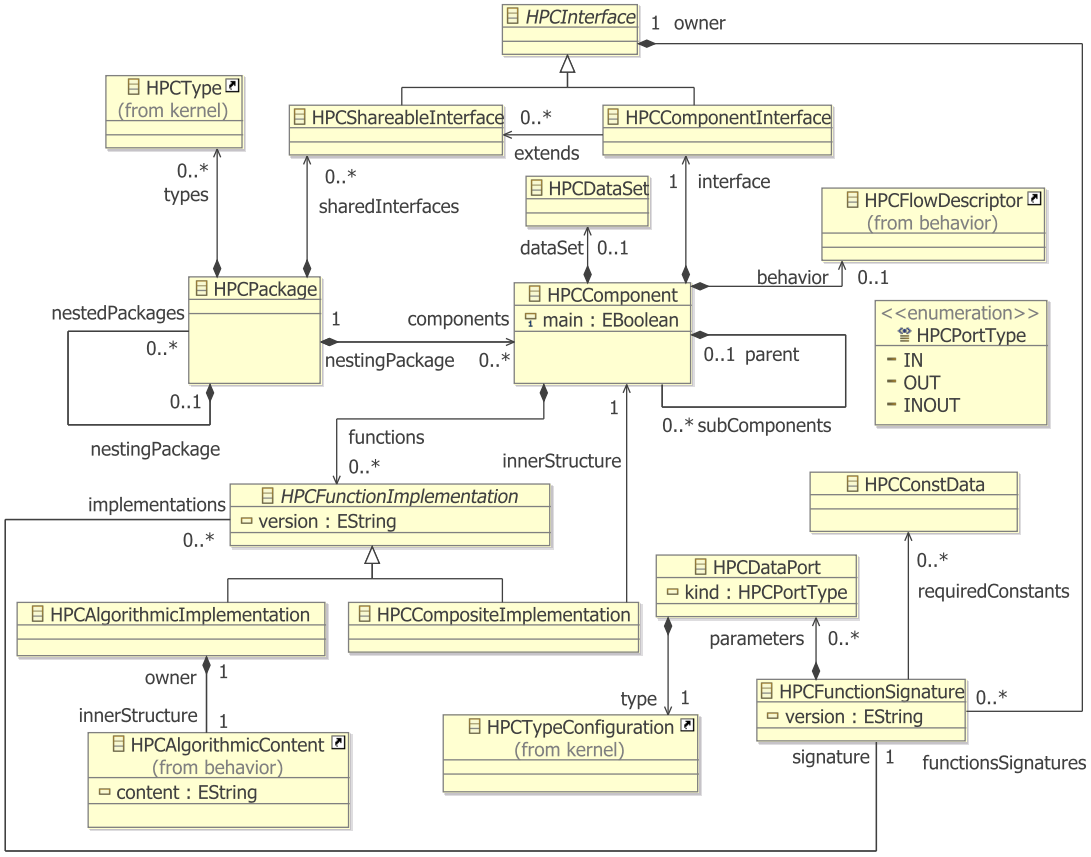


FIGURE 5.4 – Paquetage *structure* du métamodèle d'*HPCML*

Paquetage

Deux concepts permettent d'organiser les composants d'un projet, le `HPCPackage` est l'un d'entre eux. Nous verrons plus tard,lors de sa présentation, que dans certains cas, le `HPCComponent` participe aussi à la définition de la structure d'un projet. Notons aussi que les hiérarchies définies avec ces deux concepts permettent de définir l'espace de noms d'un élément de modèle avec la syntaxe « :: » pour séparer les niveaux. Par exemple, l'identifiant d'un `HPCFlowDescriptor` ressemblerait à :

```
PackageL::PackageO::ComponentS::ComponentT::HPCFlowdescriptor
```

Concentrons-nous maintenant sur le concept de `HPCPackage`. Son rôle est à la fois de contenir et d'organiser des éléments de modèle : des `HPCComponents`, des `HPCTypes`, des `HPCShareableInterfaces` mais aussi des `HPCPackage`. La définition d'un hiérarchie de paquetages est possible grâce à la relation `nestedPackages/nestingPackage`. Au passage notons que la bibliothèque de modèles qui contient tous les types de base est en fait un `HPCPackage` ne contenant que des `HPCTypes`.

Composant

Comme pour la notion de modèle, celle de composant est sujette à des définitions diverses. Dans cette thèse, nous avons choisi de raisonner à partir de la définition générale fournie par Szyperski :

> *« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »*
> C. Szyperski [200]

Le choix de définir un formalisme orienté composant a été motivé par deux raisons principales. Premièrement,les composants ont la capacité de favoriser la réutilisation et de permettre la construction d'applications par assemblage plutôt que par développement. Le calcul scientifique permet en effet de nombreuses possibilités en termes de réutilisation. Deuxièmement, les composants expriment sans ambiguïté au travers de leurs interfaces les données nécessaires et produites par les services qu'ils proposent. Cette identification claire permet de faciliter le calcul des dépendances nécessaire à la génération de code parallèle performant.

Le paquetage *structure* fournit le concept `HPCComponent` pour définir des composants. Un `HPCComponent` comprend plusieurs éléments :

- une interface (`HPCComponentInterface`) qui définit le service offert par le composant ;
- des implémentations (`HPCFunctionImplementation`) des fonctions spécifiées par l'interface du composant ;
- des constantes (`HPCDataSet`) qui sont nécessaires au fonctionnement du composant et qui sont déclarées dans l'interface du composant ;
- des sous-composants (relation `subComponents`) permettant de raffiner la spécification du composant ;
- un `HPCFlowDescriptor` permettant d'assembler et de composer des fonctions.

Nous venons de présenter le fait que les `HPCComponents` peuvent, comme les `HPCPackages`, contenir des `HPCComponents`. Quelle est alors la différence entre ces deux possibilités ? Un sous-composant permet de raffiner la spécification d'un composant parent en implémentant une partie du service de ce composant parent. De plus, un sous-composant possède une visibilité sur les constantes (`HPCDataSet`) de ses composants parents que ne possède pas un composant défini dans un paquetage. De ces différences découle la règle de modélisation suivante :


Règle de modélisation

Un composant B ne doit être défini comme sous-composant d'un composant A que si le composant B sert d'implémentation (`HPCCompositeImplementation`) à l'une des `HPCFunctionSignatures` de l'interface du composant A. Dans ce cas, la signature (relation `signature`) de l'`HPCFlowDescriptor` du composant B sera copiée à partir de l'`HPCFunctionSignature` en provenance du composant A qu'il implémente.
Dans le cas où cette condition ne serait pas remplie, le composant B pourrait être défini dans un `HPCPackage`.

Concentrons-nous maintenant sur la possibilité de définir un `HPCFlowDescriptor` dans un composant. Cette relation `behavior` est capitale dans le métamodèle d'*HPCML* puisqu'elle permet de faire la liaison entre les aspects structurels et les aspects comportementaux. Nous verrons en détails dans la section 5.1.4 en quoi consiste exactement un `HPCFlowDescriptor`. Pour le moment, considérons qu'il permet de décrire ce qu'on pourrait qualifier de « sémantique d'exécution » du composant. Par conséquent, on remarque qu'un `HPCComponent` peut posséder deux facettes différentes : d'un coté, une facette plutôt structurelle où il offre un service et de l'autre, une facette plutôt fonctionnelle dans laquelle

son exécution est possible. À partir de ce constat, on peut établir trois grandes catégories de composants :

- Les composants qui possèdent un `HPCFlowDescriptor` mais ne proposent aucun service en plus de celui de leur exécution. Dans cette catégorie, on retrouve principalement les composants servant de composant principal à un programme (propriétés `main` à vrai) ou ceux qui ne font qu'assembler des fonctions provenant d'autres composants.
- Les composants qui ne possèdent aucun `HPCFlowDescriptor` mais proposent un service défini via leur interface. On retrouve surtout dans cette catégorie les composants de premier niveau qui fournissent un service générique potentiellement utilisable par beaucoup de projets, par exemple, un service d'intégration numérique.
- Les composants qui possèdent les deux facettes. Ce sont majoritairement les composants utilisés pour raffiner un composant, c.-à-d. des sous-composants, que l'on trouve dans cette catégorie. Dans ce cas, le `HPCFlowDescriptor` peut néanmoins assembler des fonctions provenant à la fois de son composant ou d'un autre composant.


Règle de modélisation

Il est conseillé d'utiliser un nom de composant reflétant une action lorsqu'il possède une sémantique d'exécution, c.-à-d. un `HPCFlowDescriptor`. Dans le cas contraire, on choisira un nom décrivant le service offert par le composant.

Un `HPCComponent` ne possède pas d'état interne puisqu'il n'héberge que des constantes et des fonctions. C'est une propriété importante pour la parallélisation puisqu'elle garantit qu'une utilisation séquentielle de la fonction sera équivalente à une utilisation multiple en parallèle à condition, bien sûr, que la disponibilité des données d'entrées soit assurée.

Les concepts présentés dans cette section ont pour objectif de proposer un modèle de composant abstrait. Par contre, rien n'oblige à ce que le code généré à partir d'un assemblage de composants abstraits soit lui-même basé sur un formalisme orienté composant. Cette souplesse permet d'outrepasser la lourdeur d'exécution d'un assemblage de composant traditionnel et d'envisager des gains en performance lors de l'exécution.

Interface

Une `HPCInterface` permet de définir un service, c'est-à-dire une collection de signatures de fonctions (`HPCFunctionSignature`) définissant une fonctionnalité. Une signature de fonction se compose essentiellement d'un nom, puisque son concept hérite d'`HPCNamedElement`, d'un ensemble de ports de données d'entrées ou de sorties (`HPCDataPort`) et d'une liste de constantes utilisées par la fonction (`HPCConstData`). Il est intéressant de noter que le mode de gestion des constantes est clairement différent des variables traditionnelles. Nous aurons l'occasion de clarifier les raisons de cette séparation dans la section 5.1.3.

On distingue deux types d'interfaces dans le métamodèle d'*HPCML* :

- les `HPCShareableInterfaces` qui sont des interfaces rattachées à des paquetages et qui permettent de définir des services génériques que plusieurs composants pourront implémenter.
- les `HPCComponentInterfaces` qui sont des interfaces propres à des composants (`HPCComponent`) et qui permettent de définir les services offerts par ce composant. Une `HPCComponentInterface` peut étendre une ou plusieurs `HPCShareableInterfaces`

grâce à la relation `extends`. Cela signifie que le service défini par l' `HPCComponentInterface` sera l'union de l'ensemble des `HPCFunctionSignatures` des `HPCShareableInterfaces` qu'elle étend avec l'ensemble des `HPCFunctionSignatures` qu'elle définit.

Règle de *refactoring* - `HPCComponentInterface`

On souhaite définir une interface partagée (`HPCShareableInterface`) à partir d'une interface propre à un composant (`HPCComponentInterface`).

Conséquences :
- reprendre les signatures de fonctions provenant uniquement de l'interface et non des interfaces qu'elle étend ;
- dans le `HPCPackage` où se trouve le `HPCComponent` de l'interface source, créer une nouvelle `HPCShareableInterface` avec ces signatures ;
- ajouter la nouvelle interface aux interfaces étendues par l'interface source.

Implémentations

Il existe deux façons d'implémenter la signature d'une fonction (`HPCFunctionSignature`). Le choix dépend de la complexité de la fonction. Si elle est complexe, on va devoir raffiner sa spécification à l'aide d'un sous-composant via une `HPCCompositeImplementation` et son `HPCFlowDescriptor` associé (relation `innerStructure`). Par contre, si le développeur estime que la fonction à implémenter est simple alors il peut utiliser une `HPCAlgorithmicImplementation` et son `HPCAlgorithmicContent` (relation `innerStructure`). L'appréciation de la complexité d'une fonction reste à la charge du développeur. Cependant, McConnell [150], préconise de ne pas dépasser 200 lignes de code pour une fonction (hors sauts lignes et commentaires), on peut appliquer la même recommandation pour un `HPCAlgorithmicContent`. D'autre part il faut noter que l'expression de parallélisme potentiel dans un `HPCAlgorithmicContent` n'est pas formalisé pour le moment et, qu'elle ne concernera dans tous les cas que du parallélisme à grains fins. Les concepts `HPCFlowDescriptor` et `HPCAlgorithmicContent` seront présentés en détails dans la section 5.1.4.

Règle de modélisation

La possibilité de définir plusieurs implémentations (`HPCFunctionImplementation`) pour une même signature (`HPCFunctionImplementation`) doit seulement être utilisée pour définir plusieurs versions de la même implémentation (évolution, correction) à des fins de comparaison. Si le développeur souhaite proposer des implémentations différentes pour le même service il doit alors créer deux composants dont les interfaces doivent étendre une `HPCShareableInterface` contenant la définition du service.

Jeux de données

Les concepts permettant de définir un jeu de données sont importés du métamodèle *numerical* (figure 6.3) de l'outil *Paprika* que nous présenterons dans le chapitre 6.

Un composant peut posséder un jeu de données (`HPCDataSet` qui contient des `Paprika:numerical:DataBlock`). Ce dernier est un ensemble de constantes physiques, de paramètres du modèle physique et de paramètres du modèle mathématique (`Paprika:numerical:Data`). Ces constantes peuvent être référencées par les signatures du composant ou

de ses sous-composants. Comme pour les variables normales, ces données sont fortement typées et des contraintes peuvent être définies sur leurs valeurs grâce à des intervalles.

La cohérence des constantes sur l'ensemble d'un code de simulation est primordiale. Prenons un exemple caricatural avec la constante π et imaginons que la valeur de π soit définie dans une fonction par « $4.0 \times \arctan(1.0)$ » alors qu'une autre fonction possèderait aussi sa propre constante π mais initialisée à « $22.0/7.0$ ». Cette situation peut provoquer des résultats de calculs surprenants, surtout si ces deux fonctions sont amenées à utiliser une sous fonction prenant en paramètre la valeur de π. Il n'est pas évident de trouver la source de ce type d'erreur. Même s'il semble peu probable qu'un unique développeur puisse réaliser ce type de déclarations multiples, la situation reste cependant crédible en cas de réutilisation entre projets. C'est, entre autres, pour éviter ce genre de problèmes que les développeurs doivent suivre la règle de modélisation ci-dessous.


Règle de modélisation

Toutes les constantes physiques doivent être déclarées dans le jeu de données d'un composant et référencées dans les signatures des fonctions les utilisant. Leur déclaration comme variable locale est donc proscrite.

Comme nous l'avons vu dans la section 5.1.2, le respect de cette règle de modélisation est primordial afin d'obtenir un code de simulation indépendant d'unités particulières.

Un composant possède une visibilité sur son jeu de données et sur ceux de ses composants parents. Comme une application est construite à partir de plusieurs composants, il peut arriver que des constantes physiques soient déclarées dans des hiérarchies de composants indépendantes. Dans ce cas, il est nécessaire d'associer les constantes identiques lors de la transformation produisant le modèle à destination de l'outil générant l'interface graphique de création de jeux de données. Lorsque c'est possible, il est judicieux d'éviter une telle association en remontant une constante dans une hiérarchie de composants afin d'augmenter sa visibilité.

5.1.4 Paquetage *behavior*

Le paquetage *behavior* rassemble les concepts permettant de définir l'aspect comportemental d'une application. Il est possible grâce à ces concepts de construire une application en composant des fonctions fournies par les `HPCComponents` que nous avons présenté dans la section précédente. C'est aussi dans ce paquetage que l'on va retrouver les concepts permettant d'exprimer le parallélisme indépendamment d'une quelconque plateforme.

L'ensemble du contenu du paquetage *behavior* est présenté dans la figure 5.5. L'élément parent de ce paquetage est l'`HPCFlowDescriptor`. On peut le voir comme une fusion du formalisme de diagramme d'activité de *SysML* et d'*UML* car il repose à la fois sur l'expression du flot de contrôle (`HPCFlowNode`, `HPCTransition`) et sur celle du flot de données (`HPCDataPortConnector`, `HPCDataPort`).

Flot de contrôle

L'expression du flot de contrôle repose sur les concepts `HPCFlowNode` et `HPCTransition`. Ces deux concepts permettent de définir des graphes orientés où les `HPCFlowNodes` sont les nœuds et les `HPCTransitions` sont les arcs. Il existe plusieurs types d'`HPCFlowNode` possédant chacun des multiplicités de transitions différentes. Parmi les nœuds de base on retrouve :

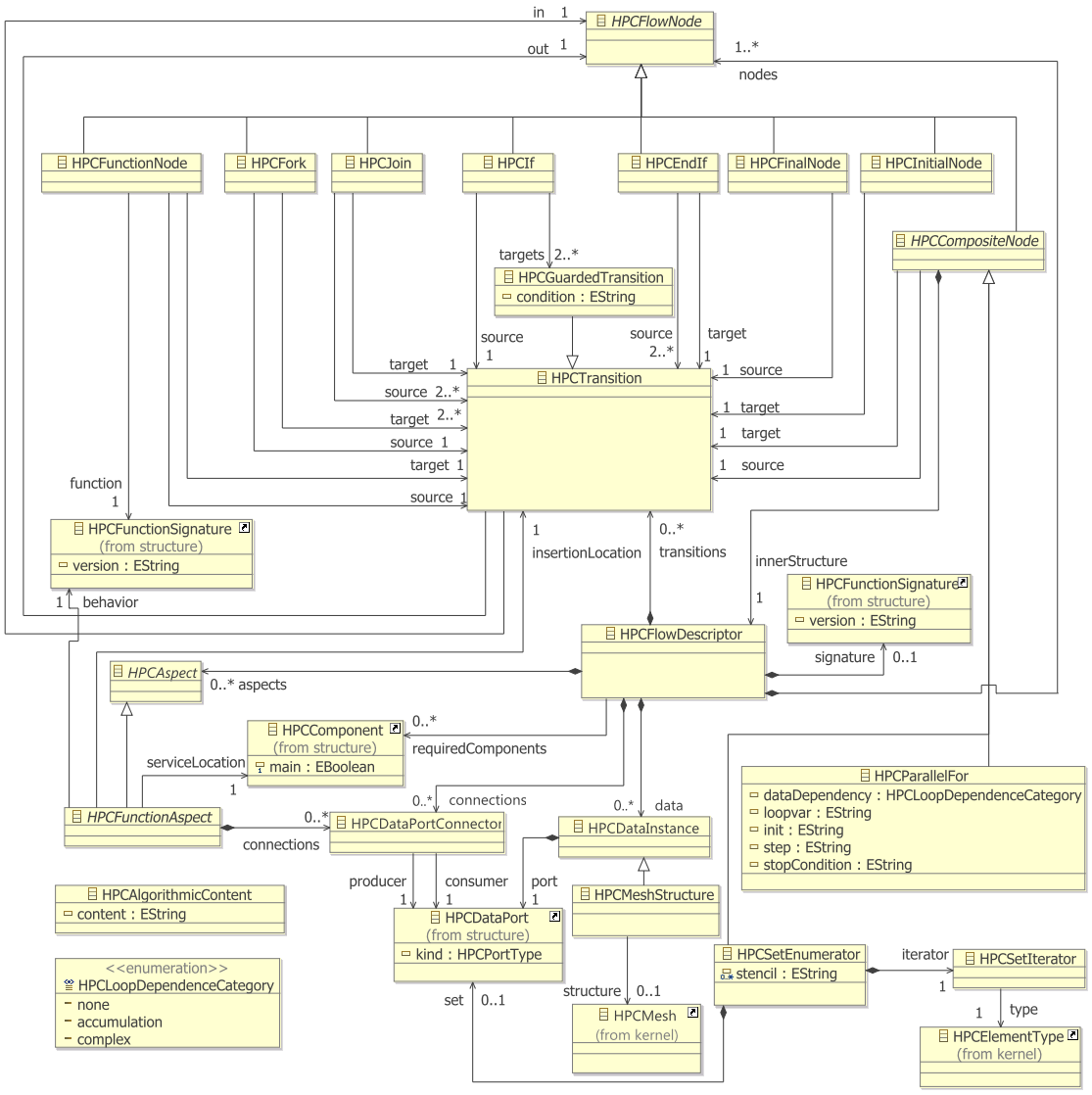


FIGURE 5.5 – Paquetage *behavior* du métamodèle d'*HPCML*

– les nœuds initiaux `HPCInitialNode` et finaux `HPCFinalNode` ;
– les nœuds permettant de définir des branchements conditionnels (`HPCIf`, `HPCEndIf`).

Composition de fonctions

L'objectif premier d'un `HPCFlowDescriptor` est de permettre la description d'algorithmes en assemblant des fonctions (`HPCFunctionSignatures`) provenant de plusieurs composants. Cette composition repose sur l'utilisation de `FunctionNode` dans le flot de contrôle. Un `FunctionNode` doit être associé avec une `HPCFunctionSignature`. De base, un `HPCFlowDescriptor` peut seulement composer des `HPCFunctionSignatures` contenues dans le composant dans lequel il est défini. Il est nécessaire de déclarer la liste des composants externes requis (relation `requiredComponents`) pour que le `HPCFlowDescriptor` puisse accéder à leurs fonctions.

L'implémentation de ce concept abstrait doit fournir le comportement suivant : la fonction associée à un `FunctionNode` sera exécutée si la transition arrivant sur le `FunctionNode`

a été exécutée et si les données demandées sur les différents `HPCDataPort` sont disponibles.

Dans le cas où une `HPCFunctionSignature` possèderait plusieurs implémentations, la liste des implémentations devrait être ajoutée au jeu de données du composant. Le développeur pourra alors décider s'il souhaite sélectionner une implémentation particulière ou laisser le choix à l'utilisateur final en propageant la liste vers l'outil chargé de la génération de l'interface graphique de mise en données.

Règle de modélisation

Afin de faciliter la compréhension, il est recommandé d'utiliser au maximum une dizaine d'`HPCFunctionNode` par `HPCFlowDescriptor` principal (c.-à-d. en comptant ceux présents dans les `HPCFlowDescriptor` contenus par les `HPCCompositeNode` du `HPCFlowDescriptor` principal).

En effet selon Miller [156], une des premières barrières limitant notre compréhension des choses, est la « faible » capacité de notre mémoire à court terme qui ne peut stocker que sept plus ou moins deux éléments d'information. En cas de `HPCFunctionNode` trop nombreux, il est recommandé de décomposer le problème en utilisant les `HPCFlowDescriptors` d'autres `HPCComponents`. Selon Siau [190], cette façon de procéder aide à outrepasser les limites évoquées.

Création et gestion des flots de données

Comme nous l'avons déjà évoqué, un `HPCFlowDescriptor` permet aussi de spécifier le flot de données d'un assemblage de `HPCFunctionSignatures`. C'est le concept `HPCDataPortConnector` qui permet de connecter des `HPCDataPorts`. Plusieurs `HPCDataPortConnectors` peuvent partir d'un même `HPCDataPort` (relation `producer`) mais un seul `HPCDataPortConnector` peut arriver dans un `HPCDataPort` (relation `consumer`).

Au sein d'un `HPCFlowDescriptor`, il existe des `HPCDataPorts` (donc susceptibles d'être connectés via un `HPCDataPortConnector`) rattachés à trois catégories de données sources :

- les variables du jeu de données du composant contenant l'`HPCFlowDescriptor` ;
- les paramètres de la signature fonctionnelle de l'`HPCFlowDescriptor` (relation `signature`) ;
- les variables définies localement grâce au concept `HPCDataInstance`. La déclaration d'une structure de données sur un maillage repose sur ce principe (`HPCMesh`) à la différence qu'il faut en plus spécifier quel maillage est concerné (relation `meshRelation`).

Expression du parallélisme

On aborde ici l'un des points les plus critiques d'*HPCML*, l'expression du parallélisme. Nous avons décidé d'abstraire le parallélisme en s'inspirant de la philosophie des patrons de conception parallèles et des squelettes algorithmiques que nous avons présentés dans la section 3.2.1. L'idée est de proposer des concepts qui représentent une stratégie de parallélisme. L'implémentation de ces stratégies parallèles pour une plateforme matérielle donnée est à la charge de l'ingénieur logiciel. Cette partie du métamodèle d'*HPCML* est donc sujette à de futures évolutions visant à rajouter de nouvelles stratégies. Pour le moment, le paquetage *behavior* propose trois façons d'exprimer le parallélisme.

Flots d'exécution concurrents La première façon d'exprimer du parallélisme est de spécifier plusieurs flots d'exécution concurrents à l'aide des concepts `HPCFork` et `HPCJoin`. La figure 5.6 montre un exemple d'utilisation de ces concepts.

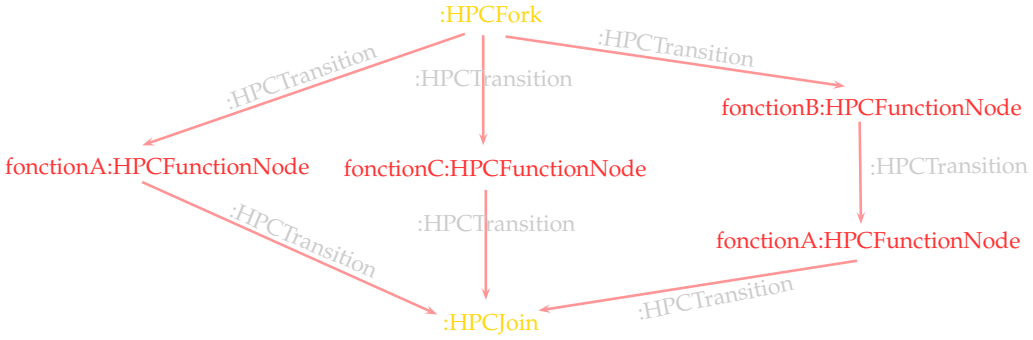


FIGURE 5.6 – Exemple d'utilisation des concepts `HPCFork` et `HPCJoin` pour l'expression de parallélisme de tâches

Le concept `HPCFork` permet d'indiquer que l'on souhaite créer plusieurs flots d'exécution à partir du flot d'exécution courant. Il est ensuite possible de décrire le contenu des flots d'exécution parallèles qui peuvent contenir des fonctions différentes ou des fonctions identiques avec des paramètres différents. Finalement, le concept `HPCJoin` permet d'indiquer que l'on souhaite synchroniser tous les flots d'exécution pour les fusionner. Les transferts de données ne sont pas représentés dans l'exemple de la figure 5.6. Or, ces transferts sont parfois problématiques car ils peuvent causer des inter-blocages entre deux flots d'exécution qui attendraient chacun de l'autre des données pour continuer leur exécution. Dans les cas où l'analyse statique ne serait pas en mesure de garantir l'absence d'inter-blocages potentiels, des erreurs ou au moins des avertissements devront être communiqués au développeur.

Si on fait le rapprochement avec les patrons identifiés par Mattson et coll [149], cette première façon d'exprimer le parallélisme correspond au *event-based coordination pattern*.

Décomposition de domaine La deuxième façon d'exprimer du parallélisme repose sur l'utilisation d'un nœud composite `HPCSetEnumerator` qui, comme son nom l'indique (*nomen est omen*), permet de parcourir tous les éléments d'un ensemble (principalement des groupes de mailles mais il y a la possibilité de définir d'autres types d'ensemble). Ce concept exploite le parallélisme en utilisant une approche par décomposition de domaine dont le principe est expliqué dans la figure 5.7 pour un maillage 1D sans perte de généralité. Le domaine de calcul, c'est-à-dire le groupe de mailles (indiqué par la relation `set`) est découpé en sous-domaines. Chaque sous-domaine est assigné à une unité d'exécution différente qui va réaliser le calcul (`HPCFlowDescriptor` de la relation `innerStructure`) sur son domaine.

Pour certains schémas numériques utilisés dans le modèle de résolution mathématique, le calcul réalisé sur une maille nécessite l'accès aux données de certaines mailles voisines. La liste contenue dans l'attribut `stencil` sert à décrire ce schéma des dépendances de façon relative. Prenons, par exemple, un maillage 1D : si le schéma numérique a besoin d'accéder à la maille de gauche et celle de droite, la liste `stencil` contiendra deux éléments : -1 et 1. Pour les dimensions supérieures, des virgules servent à séparer les coordonnées. Lorsqu'une maille a besoin d'accéder à toutes les autres mailles, il faut alors créer dans la liste `stencil` un seul élément contenant la valeur « all ». La description de ce schéma des dépendances peut s'avérer utile pour la génération de code vers certaines architectures. Par exemple,

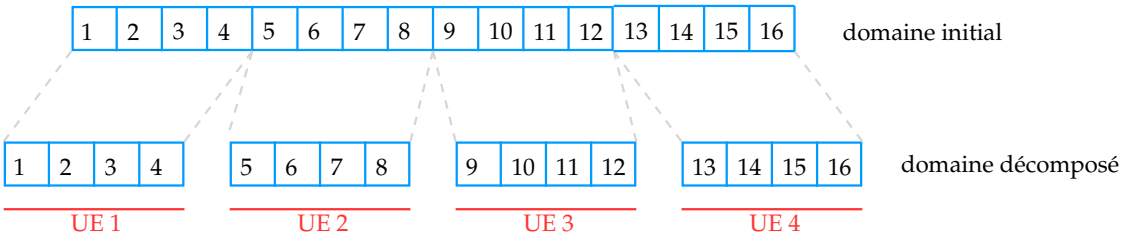


FIGURE 5.7 – Principe de la décomposition de domaine avec un maillage 1D

il est fréquent d'utiliser des mailles fantômes pour minimiser les communications sur les architectures à mémoire distribuée. Ces mailles fantômes sont des recopies des mailles du bord des sous-domaines adjacents afin de donner l'illusion que le domaine de calcul est continu. Elles permettent de regrouper les communications puisque la synchronisation ne se fait qu'une fois le calcul terminé sur le sous-domaine. Par contre, pour que cette technique fonctionne, il faut que la couche de mailles fantômes soit suffisamment large par rapport aux accès indiqués dans le schéma numérique. Le schéma des dépendances peut servir à déterminer l'épaisseur optimale de cette couche de mailles.

Si l'on continue à faire le rapprochement avec les patrons identifiés par Mattson et coll [149], cette deuxième façon d'exprimer le parallélisme correspond au *geometric decomposition pattern*.

Boucle parallèle La troisième façon d'exprimer du parallélisme repose sur le concept permettant de décrire des boucles : l'`HPCParallelFor`. Il possède une sémantique parallèle, c'est à dire qu'il essaye d'exécuter les itérations en parallèle lorsque c'est possible. Les facteurs limitant l'exécution parallèle des itérations d'une boucle sont les dépendances de données entre ces itérations. Ces dépendances de données peuvent être plus ou moins complexes ; trois catégories sont identifiées (`HPCLoopDependenceCategory`) :

- *none* : les itérations sont indépendantes mais peuvent nécessiter l'accès aux mêmes données.
- *accumulation* : chaque itération écrit dans une même variable, la dépendance peut être levée en utilisant une opération de réduction qui va recombiner le résultat de chaque itération.
- *complex* : les itérations nécessitent le résultat d'autres itérations. La parallélisation semble difficilement envisageable.

Cette troisième façon d'exprimer le parallélisme correspond au *task parallelism pattern* des patrons identifiés par Mattson et coll [149].

Gestion des aspects

Afin de ne pas mélanger les préoccupations dans un `HPCFlowDescriptor`, il est possible de spécifier autour de ce dernier, via ce que nous appelons des « aspects », les préoccupations annexes à l'implémentation du modèle mathématique de résolution. Cette approche permet de ne pas altérer la description des flots d'exécution et de données. Deux concepts abstraits pour la définition d'aspect sont présents dans le paquetage *behavior* : un concept générique (`HPCAspect`) et un concept permettant de définir des aspects appelant une fonction (`HPCFunctionAspect`).

La version présentée ici du métamodèle d'*HPCML* fournit des types d'aspects pour la gestion des sorties de données et pour la validation des applications. Ces deux possibilités

seront abordées respectivement dans la section 5.1.5 pour les sorties et dans la section 5.1.6 pour la validation.

Algorithmique de bas niveau

On trouve, dans le paquetage *behavior*, le concept d'`HPCAlgorithmicContent` qui ne possède pas de lien avec le concept d'`HPCFlowDescriptor`. Ce concept permet au développeur de décrire l'algorithmique de bas niveau via une syntaxe concrète textuelle. Si l'on regarde la structure d'une application modélisée avec le métamodèle d'*HPCML*, on observe un arbre ou les feuilles sont des `HPCAlgorithmicContent`. En effet lorsque le développeur choisit de modéliser l'implémentation d'une `HPCFonction` avec un `HPCAlgorithmicContent` plutôt qu'avec un sous-composant et son `HPCFlowDescriptor`, il stoppe les possibilités de raffinement. Le contenu d'un `HPCAlgorithmicContent` doit répondre à un problème précis.

Règle de modélisation

La définition d'un `HPCAlgorithmicContent` suppose qu'on ne souhaite plus exprimer à ce niveau de détails le parallélisme ou traiter des préoccupations annexes (sorties, validation).

La définition d'un `HPCAlgorithmicContent` n'a pas été formalisée pour le moment. Il est donc possible de réutiliser des langages existants pour cette tâche. Cependant, pour être compatible avec le métamodèle d'*HPCML* et l'esprit de l'approche *MDE4HPC*, le langage sélectionné doit posséder certaines caractéristiques :

- il doit être proche du formalisme mathématique. Par conséquent il doit gérer les entiers, les réels et les complexes ;
- il doit gérer les vecteurs et les matrices et posséder des opérateurs de calcul vectoriel et matriciel ;
- il doit posséder des bibliothèques de fonctions mathématiques de base ;
- il doit gérer la définition et la manipulation de maillages de façon abstraite comme l'a identifié Berti dans ses travaux [34] (concepts pour les éléments de maillage, itérateurs abstraits).

Le langage fourni par la plateforme *Arcane* (section 3.1.12) répond à ces critères. C'est celui que nous avons choisi pour l'outil *ArchiMDE*. L'intégration de ce langage avec les modèles sera discutée plus tard dans la section 6.4.

5.1.5 Paquetage *output*

L'objectif d'un code de simulation est de produire des résultats permettant d'analyser et comprendre le système modélisé. Le paquetage offre des concepts pour spécifier des configurations de résultats ou, autrement dit, les données de calcul qui doivent être sauvegardées pour analyse. Le contenu de ce paquetage est présenté dans la figure 5.8.

Il existe deux types de configuration de résultats pouvant être spécifiés :

- les configurations simples (`HPCSimpleOutputConfiguration`) qui permettent de demander la sauvegarde d'une seule donnée (relation `affectedDataPort` sur un port `OUT` ou `INOUT`) lors d'une exécution de fonction (relation `affectedNode`) vers une ressource particulière (relation `outputResource`) ;
- les configurations complexes (`HPCSimpleOutputConfiguration`) qui permettant d'appeler, au cours de l'exécution d'une transition (relation `insertionLocation`), des fonctions de sorties définies dans des composants externes. Les données sortant des

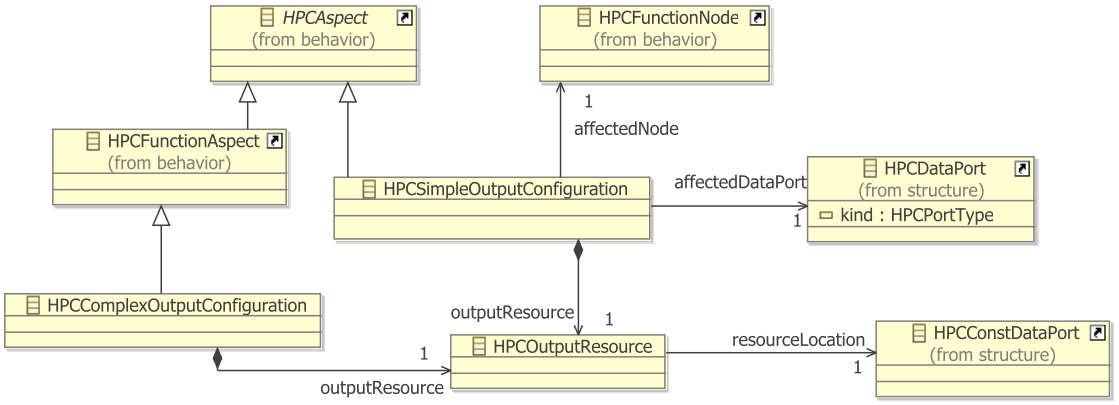


FIGURE 5.8 – Paquetage *output* du métamodèle d'*HPCML*

ports de sorties de ce type de fonctions seront sauvegardées vers une ressource particulière (relation `outputResource`). On peut d'ailleurs imaginer fournir par défaut des composants proposant des services de sorties basiques.

5.1.6 Paquetage *validation*

Le paquetage *validation* propose des concepts définissant deux approches différentes pour la mise en place de test de validation. Le contenu de ce paquetage est présenté dans la figure 5.9.

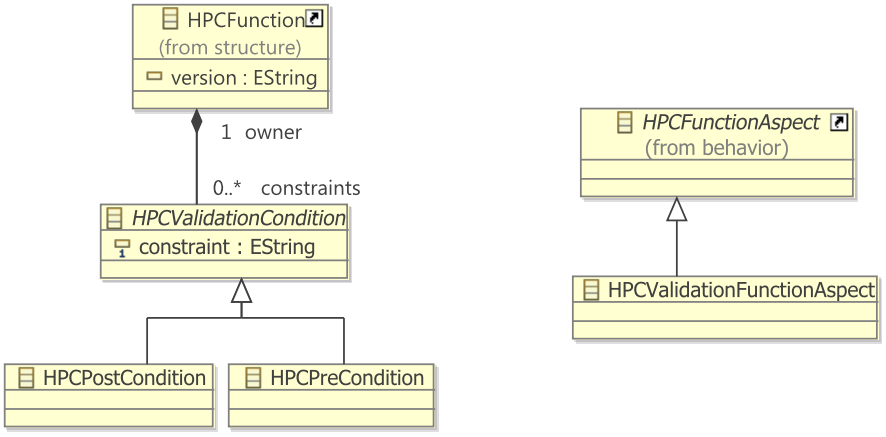


FIGURE 5.9 – Paquetage *validation* du métamodèle d'*HPCML*

La première approche repose, comme pour la définition de sorties complexes, sur le principe des aspects. Il est donc possible de définir une fonction de test (`HPCValidationFunctionAspect`) et de la rattacher à l'exécution d'une transition (relation `insertionLocation`) d'un `HPCFlowDescriptor`.

La deuxième approche s'inspire de la conception par contrat [155, 124]. Les contrats, qui sont définis par des expressions booléennes, permettent soit de spécifier les conditions d'appel d'une méthode (`HPCPreCondition`) soit de donner des informations sur ces sorties (`HPCPostCondition`). On peut considérer les pré et post-conditions comme étant respectivement un moyen décrire les droits et devoirs d'une `HPCFunctionSignature`. Dans notre

cas, nous utilisons le langage *OCL* présenté dans la section 4.1.2 pour la description de ces contrats.

Les travaux de Baudry [25] ont permis de montrer que la conception par contrat permet d'améliorer la robustesse et la diagnosabilité d'un assemblage de composant. Malgré ces bénéfices indéniables, il est difficile d'estimer l'accueil et le coût d'apprentissage du langage *OCL* par une population de numériciens.

5.1.7 Paquetage *parametric*

Le paquetage *parametric* regroupe des concepts permettant de définir des études paramétriques. Une étude paramétrique consiste à exécuter une simulation en faisant varier un ou plusieurs de ses paramètres ou fonctions avec pour objectif d'étudier plusieurs configurations ou pour déterminer des combinaisons optimales en termes de représentation de la réalité. Le contenu du paquetage *parametric* est présenté dans la figure 5.10.

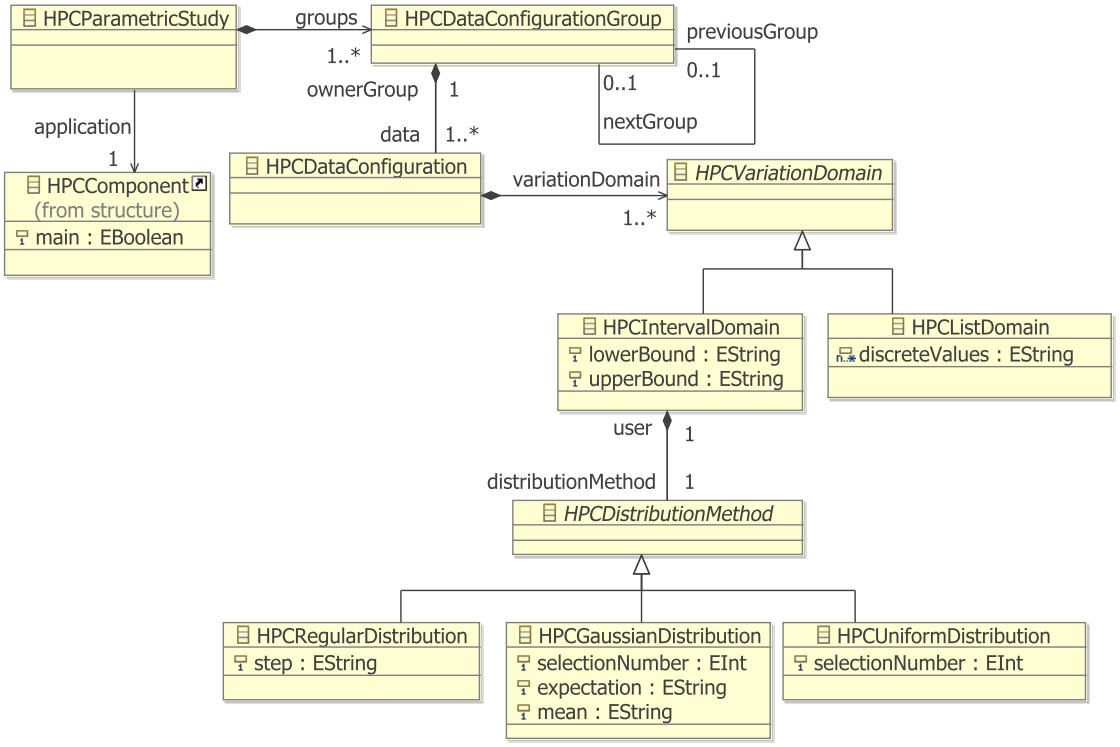


FIGURE 5.10 – Paquetage *parametric* du métamodèle d'*HPCML*

`HPCParametricStudy` est le concept parent permettant de définir un telle étude paramétrique sur une application (`HPCComponent` avec la propriété `main` à vrai). La version du paquetage *parametric* présentée ici permet uniquement de définir des variations de paramètres grâce au concept `HPCDataConfiguration`.

La définition d'un `HPCDataConfiguration` consiste à spécifier, d'une part, quel paramètre doit varier (relation `data`) et, d'autre part, de quelle manière ce paramètre doit varier en définissant un domaine de variation (`HPCVariationDomain`). Le domaine de variation d'un paramètre peut être un intervalle (`HPCIntervalDomain`) ou un ensemble de valeurs discrètes (`HPCIntervalDomain`). Dans le cas d'un intervalle, il est nécessaire de spécifier de quelle façon les valeurs doivent être distribuées sur ce dernier à l'aide d'une

`HPCDistributionMethod`. Les trois types de distributions proposés sont présentés dans la figure 5.11.

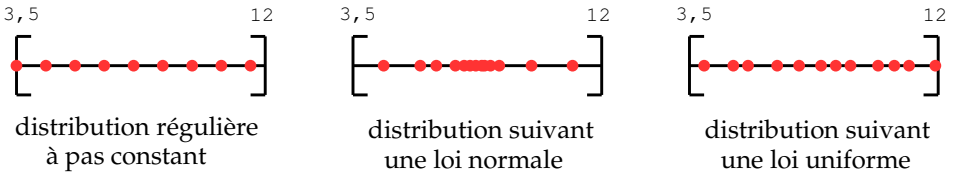


FIGURE 5.11 – Méthodes de distribution disponibles pour les études paramétriques, exemples avec l'intervalle [3,5 ; 12]

On souhaite parfois définir des scénarios de variations de paramètres, c'est-à-dire de faire varier plusieurs paramètres en même temps ou d'enchainer plusieurs variations de paramètres. Le concept `HPCDataConfigurationGroup` permet justement de décrire ce type de scénario. En effet, plusieurs `HPCDataConfiguration` se trouvant dans un `HPCDataConfigurationGroup` identique évoluent en même temps. Concrètement, à chaque itération de changement de paramètre, une nouvelle valeur de paramètre sera sélectionnée pour chaque `HPCDataConfiguration` du groupe. Cela sous entend aussi que les `HPCDataConfiguration` d'un même `HPCDataConfigurationGroup` doivent posséder un nombre de variation de paramètre identique. En addition, il est possible de décrire l'enchainement des `HPCDataConfigurationGroup` en utilisant la relation `nextGroup`.

5.2 Syntaxe concrète

La syntaxe concrète d'un langage de modélisation, c'est à dire sa représentation graphique (texte, figure, tableau, etc.), est au moins aussi importante que sa syntaxe abstraite [122]. C'est avec elle que les développeurs vont interagir. Elle joue donc un rôle clé dans l'adoption du langage. Cet aspect ne doit pas être négligé, puisque comme on l'a vu dans la section 3.2, la communauté du calcul scientifique adopte difficilement les nouvelles pratiques issues du génie logiciel.

Cette section a pour objectif de décrire la syntaxe concrète du langage *HPCML*. Nous motiverons, dans un premier temps, les choix réalisés en expliquant la démarche de conception que nous avons suivie. Dans un second temps, nous présenterons en détails la syntaxe concrète d'un diagramme d'`HPCFlowDescriptor`. Nous terminerons en donnant quelques recommandations sur la syntaxe concrète des autres aspects du langage.

5.2.1 Démarche de conception suivie

Les choix de conception concernant la syntaxe concrète du langage *HPCML* reposent essentiellement sur la théorie proposée par Moody [157] pour la conception de notations graphiques cognitivement efficaces. Selon la classification proposée par Gregor [103], c'est une théorie de type V, c'est-à-dire qu'elle indique comment concevoir un artefact (ici une notation graphique). La théorie de Moody se décompose en 9 principes que nous présentons brièvement ci-dessous :

- *clarté sémiologique* : il doit exister une correspondance 1 : 1 entre les concepts de la syntaxe abstraite et les symboles graphiques ;
- *différenciation perceptuelle* : les symboles différents doivent pouvoir être clairement différenciés ;

- *transparence sémantique* : choisir des représentations graphiques qui permettent de déduire leurs significations ;
- *gestion de la complexité* : proposer des mécanismes explicites permettant de faire face à la complexité d'un diagramme ;
- *intégration cognitive* : offrir des mécanismes explicites permettant l'intégration d'information provenant de différents diagrammes ;
- *expressivité visuelle* : utiliser toute l'étendue et la capacité des variables graphiques (voir figure 5.12) ;
- *encodage double* : utiliser du texte en complément de l'information graphique ;
- *économie graphique* : le nombre de symboles graphiques doit être cognitivement gérable ;
- *correspondance cognitive* : utiliser différent dialectes visuels pour différentes tâches et audiences.

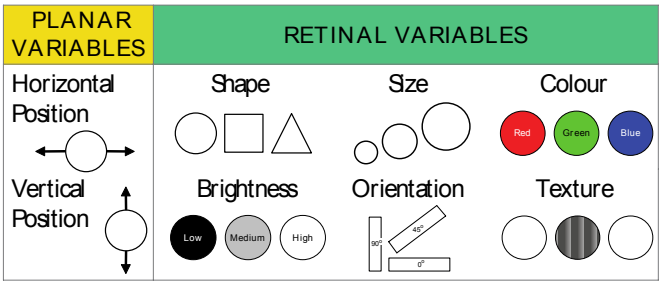


FIGURE 5.12 – Variables visuelles[157]

5.2.2 Syntaxe concrète d'un diagramme comportemental

Cette section présente la syntaxe concrète des diagrammes comportementaux qui correspondent au concept `HPCFlowDescriptor`. Nous évoquerons aussi à plusieurs reprises des fonctionnalités annexes au diagramme que doit posséder un outil implémentant cette syntaxe concrète. Cet outil devra au minimum posséder les zones décrites dans la figure 5.13 :

- une zone pour le diagramme en lui-même ;
- une zone permettant de sélectionner le mode d'affichage. Nous verrons par la suite qu'il est, en effet, souhaitable de cacher certaines informations en fonction de l'objectif de l'utilisateur. Cette approche est destinée à faciliter la compréhension et l'ergonomie du diagramme.
- une zone permettant d'afficher la palette d'outils. La palette d'outils ne doit proposer que les concepts correspondants au mode d'affichage courant.
- une zone affichant les propriétés de l'élément de modèle sélectionné sur le diagramme.

Nous ferons par la suite référence à ces quatre zones. La couleur possède une sémantique dans les symboles de la syntaxe concrète. Elle indique à quelle facette du langage (flot d'exécution, flot de données, parallélisme, aspects) le symbole appartient. Ce choix permet à un utilisateur d'identifier plus rapidement les éléments de modèle qui l'intéressent en fonction du type de tâche qu'il doit réaliser.

Flot d'exécution

Les concepts permettant de définir le flot d'exécution possèdent une couleur rouge à l'exception de l'`HPCInitialNode` qui est vert. Cette couleur correspond mieux à l'idée de

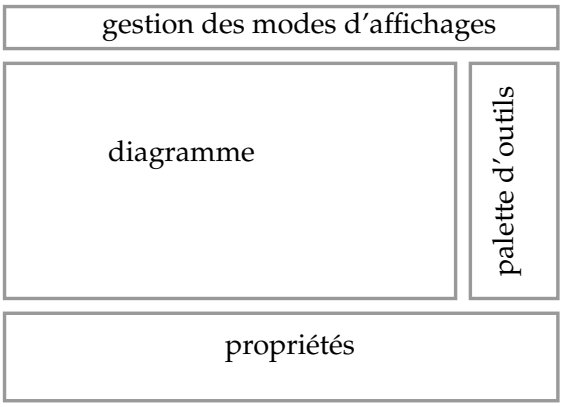


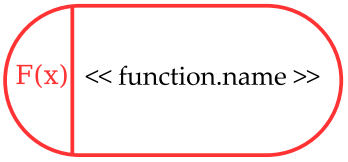
FIGURE 5.13 – Présentation des quatre zones requises pour un outil implémentant la syntaxe concrète d'un `HPCFlowDescriptor`

départ ou de commencement que la couleur rouge. De plus, cette distinction permet de rapidement identifier le début d'un flot d'exécution.

Syntaxe concrète - `behavior:HPCTransition`

En accord avec le huitième principe de Moody, le concept `HPCGuardedTransition` qui hérite de `HPCTransition` possède le même symbole que son concept parent. Par contre, le symbole représentant un `HPCGuardedTransition` sera accompagné de la condition de branchement de la transition. Elle sera écrite entre crochets : [`condition`].

Syntaxe concrète - `behavior:HPCFunctionNode`



Le concept `HPCFunctionNode` est, sans nul doute, le concept le plus important du diagramme vis-à-vis de la compréhension globale du comportement décrit. C'est pourquoi nous avons choisi une forme aux lignes arrondies car les gens ont tendance à préférer ce style de forme plutôt que des formes avec des angles comme les rectangles [19]. Nous avons aussi rajouté le symbole « $f(x)$ » (principes 2, 3, 6 et 7 proposés par Moody).

Alors qu'un simple clic devra afficher les propriétés de la fonction rattachée au `HPCFunctionNode`, un double clic devra afficher l'implémentation de la fonction. On obtiendra donc soit un diagramme `HPCFlowDescriptor`, soit le contenu d'un `HPCAlgorithmicContent`.

Comme une application est un assemblage de composants il n'est pas possible de proposer la fonctionnalité inverse, c'est-à-dire de remonter au composant parent puisque plusieurs composants peuvent utiliser le service proposé par un composant. Par contre, on pourra

stocker temporairement l'historique de parcours de la hiérarchie d'assemblage afin de proposer, dans la zone de gestion des modes d'affichages, une fonctionnalité ressemblante. Un tel mécanisme de navigation permet à l'utilisateur de découvrir la décomposition fonctionnelle d'une application sans se soucier de la façon dont les composants ont été assemblés.

Syntaxe concrète - `behavior:HPCInitialNode`

L'utilisation de la couleur verte en conjonction avec un triangle isocèle orienté vers la droite vise une bonne transparence sémantique en évoquant le sigle « jouer » que l'on retrouve sur de nombreux boutons pour évoquer le commencement de quelque chose.

Syntaxe concrète - `behavior:HPCFinalNode`

Pour les mêmes raisons que celles du symbole de l'`HPCInitialNode`, le symbole de l'`HPCInitialNode` rappelle le symbole rencontré fréquemment sur des boutons qui évoque la « fin » ou l'« arrêt » de quelque chose.

Syntaxe concrète - `behavior:HPCIf`

Le symbole de branchement conditionnel (`HPCIf`) est inspiré du losange que l'on retrouve dans beaucoup de langages de flots. Par contre, la principe de l'encodage double a été appliqué, d'où l'ajout d'un point d'interrogation évoquant une zone de décision.

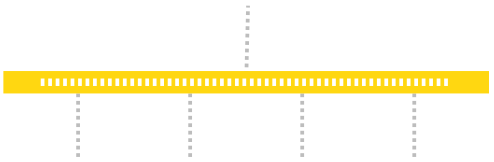
Syntaxe concrète - `behavior:HPCEndIf`

La forme du symbole de l'`HPCEndIf` provient de la forme d'un entonnoir qui véhicule cette idée de rassemblement.

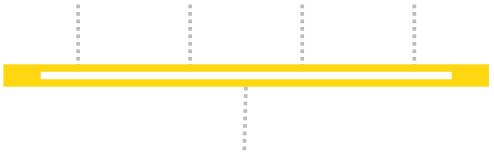
Parallélisme

Les concepts permettant d'exprimer le parallélisme sont de couleur orange.

Syntaxe concrète - `behavior:HPCFork`



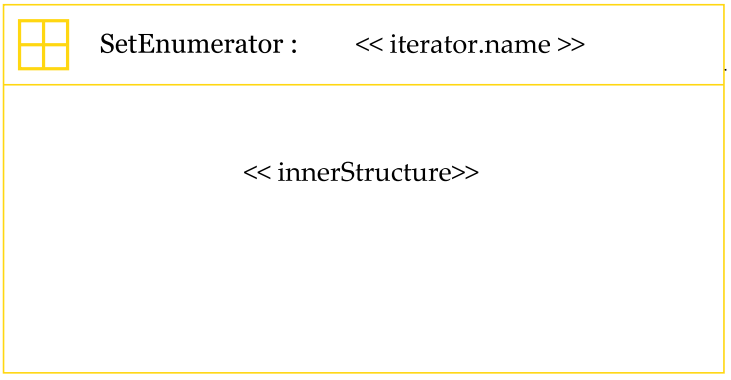
Syntaxe concrète - `behavior:HPCJoin`



Les symboles des concepts `HPCFork` et `HPCJoin` sont tous deux inspirés de traits évoquant une barrière. Comme pour le nœud décisionnel, ce symbole est fréquemment présent dans d'autres langages de modélisation. À la différence de ces langages qui utilisent le plus souvent le même symbole pour deux concepts, nous avons choisi de les différencier tout en améliorant leur transparence sémantique (principes 2 et 3).

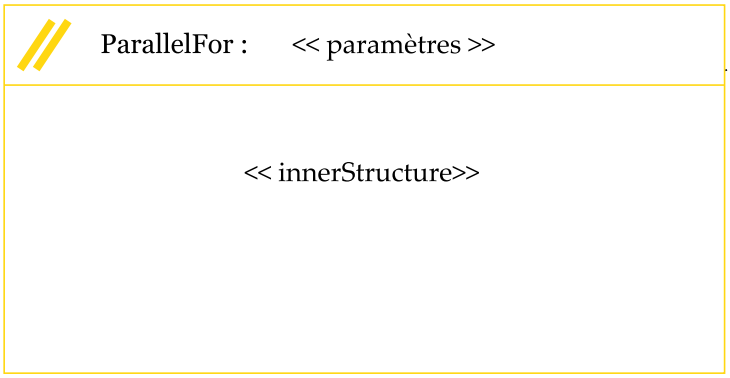
L'intérieur du symbole représentant un `HPCFork` se compose d'une multitude de traits verticaux évoquant la sortie de plusieurs flots. A l'opposé, l'intérieur du symbole représentant un `HPCJoin` se compose d'un unique bloc évoquant la sortie d'un seul flot.

Syntaxe concrète - `behavior:HPCSetEnumerator`



La forme rectangulaire de la représentation de l'`HPCSetEnumerator` a pour objectif de rappeler la zone de diagramme puisqu'un `HPCSetEnumerator` va contenir lui aussi un `HPCFlowDescriptor`. L'icône se situant en haut à gauche représente la décomposition d'un domaine (le grand carré) en quatre sous-domaines.

Syntaxe concrète - `behavior:HPCParallelFor`



La représentation de l'`HPCParallelFor` est proche de celle de l'`HPCSetEnumerator` car leur sémantique n'est pas très éloignée et qu'ils possèdent un concept parent en commun. Ils se différencient toutefois par leurs icônes. Celle de l'`HPCParallelFor` est composée de deux traits parallèles qui évoquent la sémantique parallèle du concept.

Flot de données

Afin de ne pas surcharger le diagramme, les concepts concernant la définition des flots de données ne sont pas affichés initialement sur un diagramme. Il faut demander leur affichage dans la zone de gestion des modes d'affichages. Les concepts de cette catégorie sont de couleur bleu.

Syntaxe concrète - `structure:HPCDataPort IN`

Syntaxe concrète - `structure:HPCDataPort OUT`

Syntaxe concrète - `structure:HPCDataPort INOUT`

La représentation de l'`HPCDataPort` varie en fonction de son type (propriété `kind`). Une forme triangulaire est utilisée à l'intérieur du symbole pour exprimer le sens du flot de données grâce à son orientation : triangle pointant vers l'extérieur (`OUT`), triangle pointant vers l'intérieur (`IN`), les deux triangles (`INOUT`).

Lorsqu'un `HPCFlowDescriptor` possède une `HPCFunctionSignature` les ports de cette dernière seront affichés sur un des bords de la zone diagramme lorsque le mode d'affichage correspondant aux données est activé.

Syntaxe concrète - `behavior:HPCDataPortConnector`

Le symbole de l'`HPCDataPortConnector` est composé de deux traits bleus parallèles faisant penser à un canal fluvial. On imagine alors naturellement le flot de données couler à travers.

À noter que nous avons choisi de ne pas représenter les `HPCDataInstances` dans la zone diagramme afin de ne pas surcharger celle-ci à l'extrême. Les `HPCDataInstances` d'un `HPCFlowDescriptor` doivent être accessibles dans la vue propriétés lorsque l'utilisateur clique sur celui-ci (c'est-à-dire à n'importe quel endroit du diagramme qui ne contient pas d'autres types d'éléments).

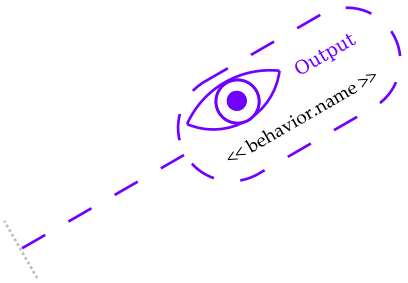
Aspects

Comme pour les flots de données, les aspects doivent être masqués lors de l'affichage initial du diagramme afin de réduire la complexité de ce dernier. La zone de gestion des modes d'affichage doit permettre d'activer l'affichage d'une famille d'aspect (sorties, validation). Les aspects de type *output* sont en violet (lien avec la couleur bleu des flots de données) et les aspects de type *validation* sont en vert.

Syntaxe concrète - `validation:HPCValidationFunctionAspect`

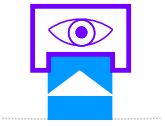
Le concept `HPCValidationFunctionAspect` hérite de `HPCFunctionAspect` qui possède un comportement proche d'un `HPCFunctionNode` mais qui se différencie par son point d'attache à une transition plutôt que dans le flot d'exécution. C'est pourquoi la forme du symbole de l'`HPCValidationFunctionAspect` a été copiée sur celle de l'`HPCFunctionAspect`. Mis à part cette origine commune, les deux symboles se différencient en plusieurs points : orientation (ici 30 degrés), le contour de la forme en pointillé, l'icône incluant un v évoquant la « vérification » ou la « validation ». En accord avec le principe 7 du double encodage, la catégorie de l'aspect est inscrite à droite de l'icône.

Syntaxe concrète - `output:HPCComplexOutputConfiguration`



Le symbole de l'`HPCComplexOutputConfiguration` est proche du symbole de l'`HPCValidationFunctionAspect` car ils héritent du même concept et interagissent donc de la même façon avec les autres éléments du diagramme. Les deux symboles se différencient toutefois par l'icône qui représente un œil, le texte et la couleur de la catégorie de l'aspect.

Syntaxe concrète - `output:HPCSimpleOutputConfiguration`



La représentation de l'`HPCSimpleOutputConfiguration` vient se greffer sur l'`HPCDataPort` auquel il est rattaché. Son symbole représente un connecteur venant s'enficher telle une prise dans le port. Cette forme vise à faire penser que le connecteur va accéder les données sortant du port. L'icône d'un œil (identique à celle de l'`HPCComplexOutputConfiguration`) symbolise que les données captées seront stockées pour être visualisées.

Syntaxe simplifiée

Il est intéressant de noter que la syntaxe concrète des diagrammes comportementaux a été conçue de façon à être utilisée au travers de logiciels de modélisation ou lors de sessions informelles de modélisation sur une feuille de papier ou un tableau. Par exemple, la couleur est utilisée pour mettre en avant les différents groupes de concepts (contrôle, données, parallélisme, etc.) mais elle n'est jamais la seule variable visuelle permettant de différencier la représentation de deux concepts. Par conséquent, un diagramme réalisé en noir et blanc à l'aide d'un stylo ou d'un marqueur ne permet la confusion de deux concepts.

5.2.3 Syntaxe concrète de la partie structurelle

Cette section présente la syntaxe concrète de la partie structurelle qui est un `HPCPackage` à sa racine. La syntaxe concrète de cette partie n'est pas un diagramme mais respecte sur le paradigme d'interface « *Master/Details* » présenté dans la figure 5.14.

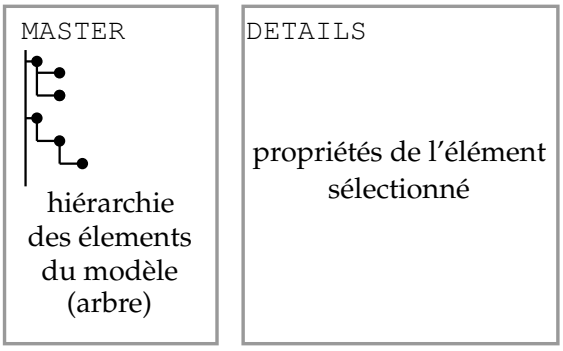


FIGURE 5.14 – Présentation des deux zones requises pour un outil implémentant la syntaxe concrète d'un `HPCPackage`

Le paradigme d'interface « *Master/Details* » repose sur deux zones :
- une zone *Master* située à gauche et contenant la hiérarchie des éléments du modèle. Ces éléments sont affichés sous forme d'un arbre reprenant l'organisation définie par la syntaxe abstraite. Un nœud de l'arbre affiche une icône, le nom du concept et le nom de l'élément de modèle. La zone *Master* doit gérer les fonctionnalités de copier ou coller pour les éléments de modèle.
- une zone *Details* affichant les propriétés de l'élément sélectionné dans la zone *Master*.

5.3 Conclusion

Dans ce chapitre, nous avons présenté le langage *HPCML*$_n$ qui permet de modéliser des applications de simulation numérique haute-performance. Dans un premier temps, nous avons abordé la syntaxe abstraite du langage dont les concepts sont formalisés à l'aide d'un métamodèle divisé en six paquetages : *kernel*, *strucuture*, *behavior*, *output*, *validation*, *parametric*. Retenons que l'approche proposée par le langage *HPCML*$_n$ pour l'expression du parallélisme repose sur quatre concepts abstraits représentant des stratégies de parallélisme (paquetage *behavior*). Dans un second temps, nous avons introduit et justifié la syntaxe concrète de ce langage.

Par la suite, nous aurons l'occasion de découvrir comment le langage a été implémenté lors de la présentation de l'outil *ArchiMDE* dans le chapitre 6. Nous verrons aussi quelques exemples d'utilisation de ce langage lors de la présentation du cas d'étude sur un code d'électromagnétisme 3D dans le chapitre 7.

Troisième partie

Outillage et évaluation

Chapitre

6

ArchiMDE : un outil pour l'approche MDE4HPC

Eurêka !
Archimède.

Sommaire

Seul un véritable atelier de génie logiciel (*AGV*) regroupant l'ensemble des services nécessaires aux différents métiers prenant part au développement serait en mesure de répondre aux exigences de l'approche *MDE4HPC*. Un tel *AGV* devrait, entre autres, gérer les activités de modélisation des différents métiers, de transformation de modèles, de validation, de génération de code, de compilation, de débogage et de gestion de versions.

Ce chapitre présente l'outil *ArchiMDE* (prononcé « Archimède »), un *AGV* de cette nature qui implémente le langage $HPCML_n$ et respecte les préconisations de l'approche *MDE4HPC*. Après une présentation de son architecture logicielle et de l'outil *Paprika* qu'il intègre, nous détaillerons son processus de développement qui aboutit à la génération d'applications basées sur la plateforme *Arcane*. Finalement, nous nous attarderons sur la problématique d'implémentation de l'intégration de l'algorithmique de bas niveau avec les modèles.

6.1 Un atelier de génie logiciel basé sur la plateforme Eclipse

Dans le chapitre 4, nous avons présenté l'approche *MDE4HPC* et ses fondements théoriques qui reposent sur l'*IDM*. Or, un des avantages de l'*IDM* est qu'il existe tout un panel d'outils permettant d'implémenter des approches s'appuyant sur celui-ci. Parmi ce panel, on retrouve des outils qui implémentent plus ou moins fidèlement les standards de l'*OMG*, comme l'ensemble des projets de l'*Eclipse Modeling Project* [195], alors que d'autres proposent leur propre vision de l'*IDM*, par exemple les outils *MetaEdit+* [126] et *DSL Tools* [60]

Notre choix s'est porté sur la plateforme *Eclipse* [55] car elle possède plusieurs caractéristiques intéressantes en addition de ses prédispositions pour l'*IDM*. C'est tout d'abord une plateforme libre dont l'architecture respecte la spécification *OSGi* au travers de l'implémentation *Equinox*. La spécification du *framework* *OSGi* implémente un modèle de composants dynamique et complet pour le langage *Java*. Les composants, appelés *bundles* ou *plugins* dans l'environnement *Eclipse*, peuvent être installés, démarrés, arrêtés, mis à jour ou désinstallés de manière distante et à chaud, c'est-à-dire sans nécessiter un redémarrage. Les différents *plugins* sont informés de l'ajout ou de la suppression de nouveaux services via un répertoire commun appelé *registry*. Cette souplesse permet de facilement créer et déployer des applications en combinant des *plugins*. Depuis les débuts de la plateforme *Eclipse*, la communauté a produit une collection importante de *plugins* pour répondre à des besoins très variés. C'est grâce à cette diversité que l'on trouve à la fois des *plugins* dédiés à l'*IDM* et des *plugins* plus axés vers le calcul scientifique. On peut citer, par exemple, les outils d'édition et de compilation de code du projet *CDT* pour le langage *C++* et du projet *Photran* pour le langage *Fortran*. On trouve aussi des *plugins* dédiés au développement et à l'utilisation d'applications parallèles dans le projet *PTP* [212].

Comme nous l'avons déjà évoqué, la plateforme *Eclipse* renferme l'*Eclipse Modeling Project* qui se concentre sur l'évolution et la promotion des technologies de développement à base de modèles. En son cœur, se trouve l'*Eclipse Modeling Framework* (*EMF*) [195] qui fournit un langage de métamodélisation nommé *ECore*. *ECore* est un métamétamodèle dont la spécification se rapproche de l'*essential MOF* standardisé par l'*OMG* [108]. Le métamodèle *HPCML* est implémenté dans l'outil *AchiMDE* via le langage *ECore*.

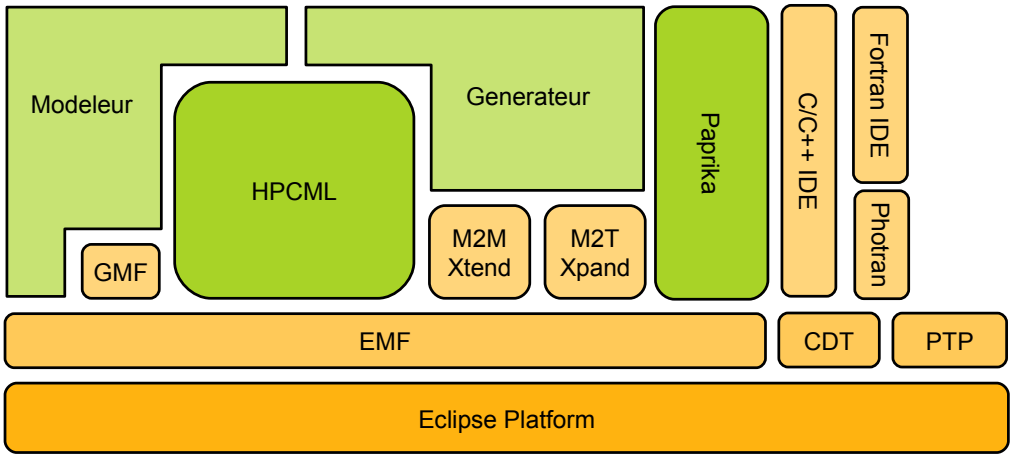


FIGURE 6.1 – Architecture logicielle de l'outil *ArchiMDE*

La figure 6.1 offre une vue simplifiée de l'architecture logicielle d'*ArchiMDE*. En addition des composants que nous venons d'aborder on retrouve des composants prenant en charge les transformations de modèles (*Xtend*, *Xpand*) ainsi que *GMF* qui fournit un approche de développement basée sur les modèles pour implémenter la syntaxe concrète d'un métamodèle. Finalement, il reste le composant *Paprika* que nous allons présenter en détails dans la section suivante.

6.2 Paprika Studio

Des travaux antérieurs à ceux présentés dans cette thèse abordant l'utilisation de l'*IDM* pour le développement de logiciels de simulation numérique ont été réalisés au travers de

l'outil *Paprika* [161] dans notre laboratoire. Cet outil a pour objectif de faciliter la création et la maintenance des interfaces graphiques dédiées à l'édition des jeux de données servant à paramétrer les codes de simulation numérique.

Les différentes étapes du développement d'une interface graphique avec l'outil *Paprika* sont présentées dans la Figure 6.2. La première étape d'un développement *Paprika* est la modélisation de la structure et des contraintes propres aux jeux de données du code de calcul visé. Cette étape se base sur le métamodèle *Numerical* qui est présenté dans la figure 6.3. Les deux concepts fondamentaux de ce métamodèle sont les types (*Type*) et les données (*AbstractData*). Des types complexes peuvent être construits en combinant ou en étendant les types simples fournis de base par *Paprika*. Une des particularités de ce métamodèle est qu'il permet d'associer un type à une grandeur physique. En ce qui concerne les données, le métamodèle *Numerical* fournit des primitives permettant de structurer ces dernières en groupes (*DataBlock*).

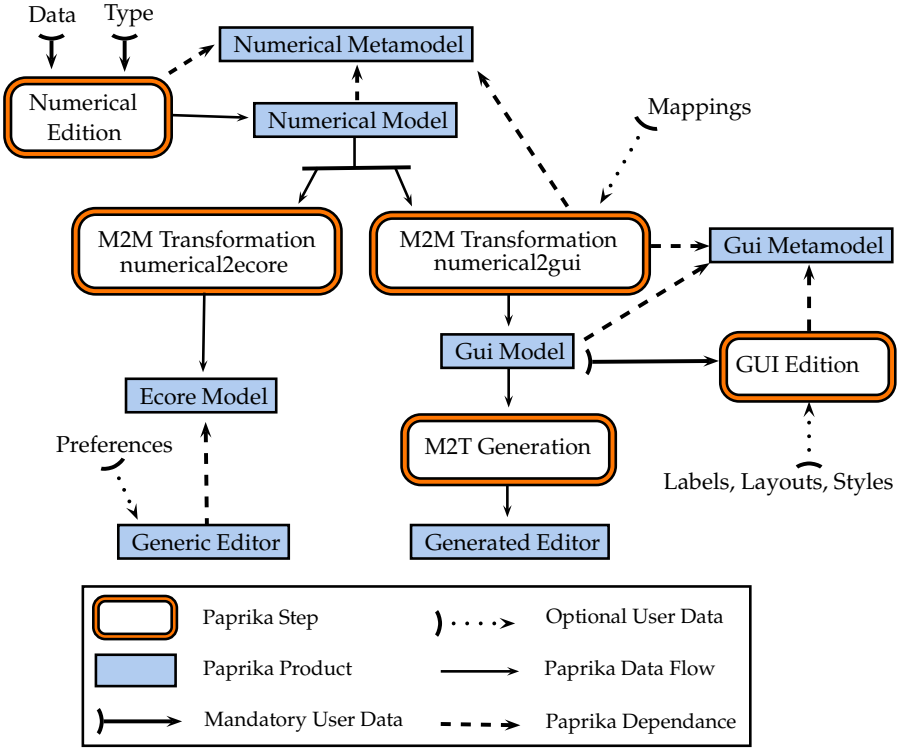


FIGURE 6.2 – Étapes d'un développement avec l'outil *Paprika* [161]

Une fois le modèle *Numerical* défini, *Paprika* est en mesure de générer ou de fournir un éditeur graphique permettant de créer et d'éditer des jeux de données respectant la structure et les contraintes définies dans ce modèle. La figure 6.2 montre clairement les deux voies possibles permettant d'obtenir un éditeur avec l'outil *Paprika* : la génération ou la généricité. Dans la suite de cette section nous détaillons ces deux approches.

6.2.1 Editeur généré

Dans ce cas, le développeur commence par définir un ensemble de règles par défaut permettant d'associer un type de données à un type de *widget* graphique (par exemple, un réel sera saisi dans un champ texte) et obtient par transformation un modèle d'interface graphique basé sur le metamodèle *HMI* (voir [161] pour de plus amples détails). Le déve-

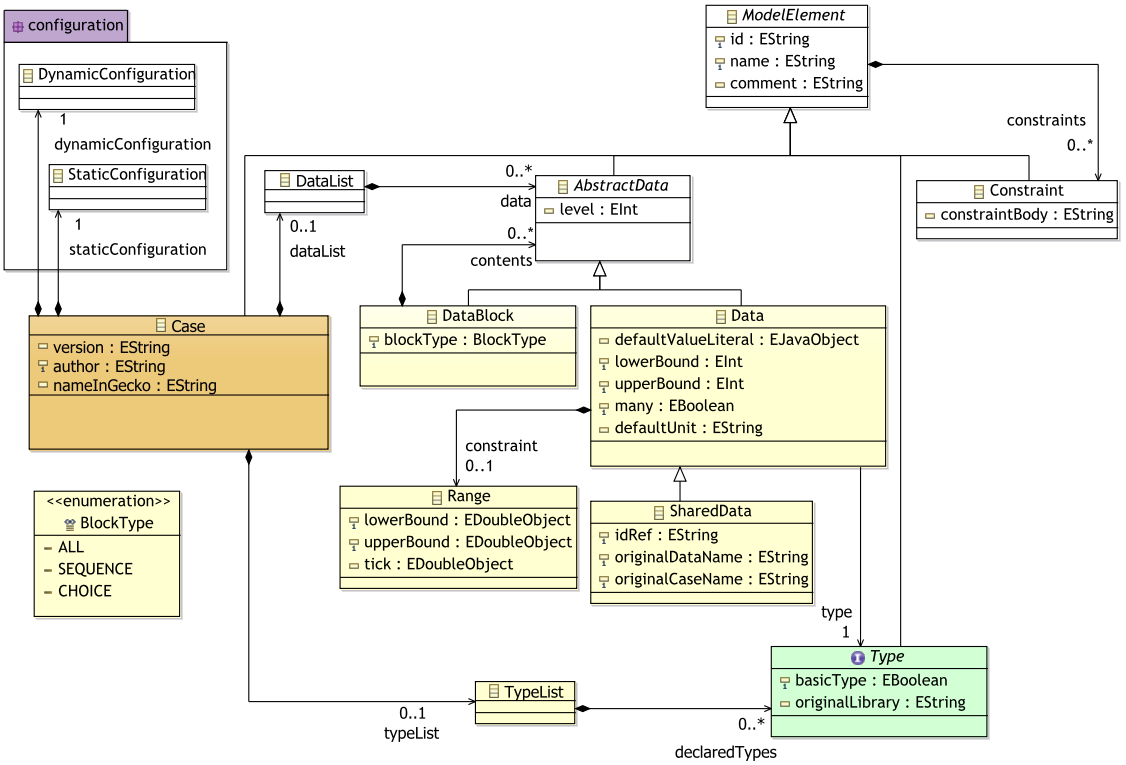


FIGURE 6.3 – Vue simplifiée du métamodèle *Numerical* de l'atelier *Paprika*

loppeur peut alors modifier le modèle généré pour répondre plus finement à ses besoins. Une fois satisfait de ses modifications, le développeur lance la génération de l'interface correspondante. *Paprika* possède à l'heure actuelle un générateur pour *GWT* [117] et pour *SWT* [164].

6.2.2 Editeur générique

Dans ce cas, le développeur choisit d'utiliser directement l'éditeur générique *Paprika* qui est en mesure de manipuler les instances de n'importe quel modèle *Numerical*. Pour ce faire, le modèle *Numerical* doit subir au préalable une transformation de modèles qui va le promouvoir en un métamodèle conforme au métamétamodèle *Ecore* et qui va injecter les particularités du modèle sous formes d'annotations compréhensibles par l'éditeur générique *Paprika*. L'éditeur générique utilise ensuite les interfaces réflectives fournies par *EMF* pour manipuler les modèles indépendamment de leur métamodèle. Ces interfaces réflectives définies au niveau du métamétamodèle *Ecore* permettent en effet de découvrir de façon générique la structure d'un modèle quelconque à l'exécution.

6.2.3 Comparaison des approches

Ces deux approches ont fait l'objet d'une comparaison détaillée dans [161]. Dans notre cas, l'approche générique a été appréciée lors du développement de l'outil *ArchiMDE* puisqu'elle est particulièrement adaptée à des évolutions fréquentes de modèles.

6.3 Fonctionnement de l'outil

Le fonctionnement de l'outil *ArchiMDE* est présenté dans la figure 6.4 au travers de l'enchainement des différentes transformations de modèles sur lesquelles il repose. Dans le chapitre 4, nous avons proposé de fournir plusieurs points de vue, aussi appelés perspectives, sur les modèles qui seraient adaptés aux différents profils métier des participants au développement d'un logiciel de simulation numérique. On retrouve cet esprit dans la version d'*ArchiMDE* présentée dans cette section.

Les numériciens profitent des perspectives permettant d'implémenter un schéma numérique le plus indépendamment possible de la plateforme :

- perspective HPCML.behavior ;
- perspective HPCML.structure ;
- perspective HPCML.validation ;
- perspective HPCML.output ;
- perspective Paprika.numerical·

Les développeurs spécialistes de l'environnement des codes bénéficient des perspectives pour la gestion des entrées/sorties d'un code de simulation :

- perspective HPCML.output ;
- perspective Paprika.numerical ;
- perspective Paprika.hmi·

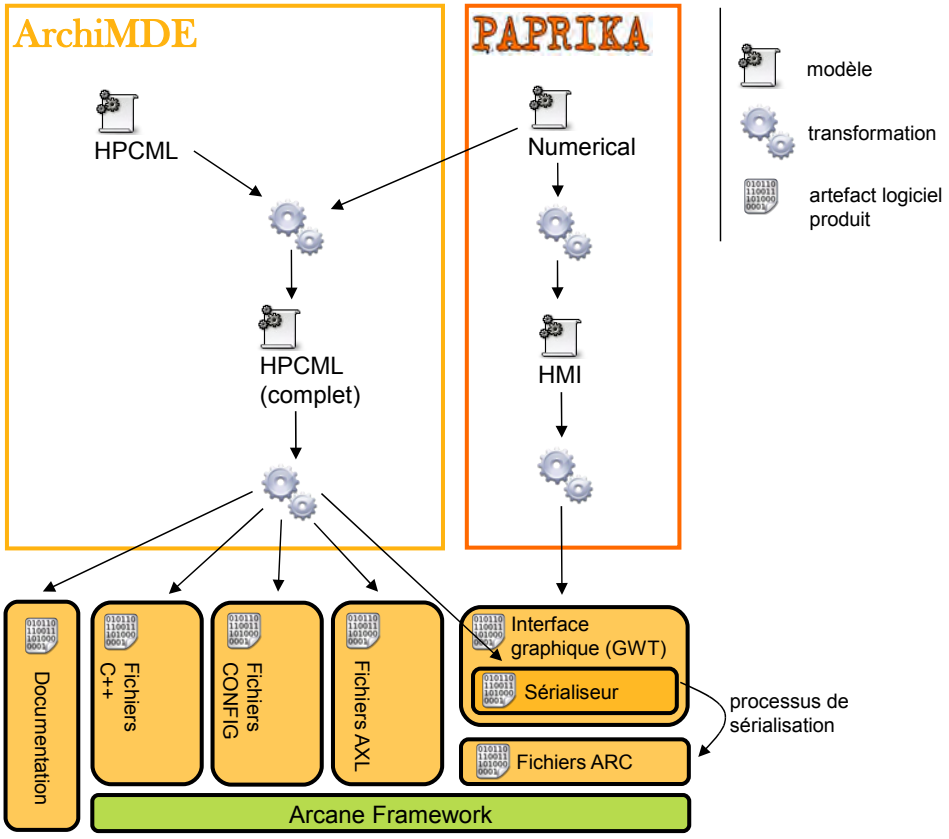


FIGURE 6.4 – Étapes d'un développement avec l'outil *ArchiMDE*

Finalement, les développeurs logiciel déterminent au travers de transformations comment les modèles abstraits conçus par les numériciens seront traduits en code exécutable pour une plateforme cible.

En ce qui concerne la plateforme logicielle cible, l'outil *ArchiMDE* est capable de générer des applications basées sur la plateforme *Arcane* que nous avons présentée dans la section 3.1.12. On retrouve dans la figure 6.4, l'ensemble des artefacts logiciels qui sont nécessaires pour créer une application *Arcane*. Il y a tout d'abord, les fichiers *AXL* qui décrivent les *modules* avec leurs différents *points d'entrée* et *variables*. Il y a ensuite les fichiers *C++* implémentant ces *modules* et les fichiers *config* décrivant comment ils sont composés entre eux. Finalement, il reste les fichiers *arc* qui sont les fichiers de jeux de données permettant de paramétrer le code de simulation. Généralement, il y a une duplication d'information entre le code source d'un code de simulation et l'interface graphique associée pour éditer ses jeux de données. Le plus souvent ces derniers étant écrits dans des langages de programmation différents, un mécanisme de sérialisation doit être mis en place afin d'établir la communication entre les deux applications. L'outil *Paprika* ne connaissant pas la façon dont un code sérialise ces jeux de données, il ne peut pas générer automatiquement le sérialiseur de l'éditeur de jeux de données. Bien que cette étape soit élémentaire, elle était donc toujours réalisée de façon manuelle. Or *ArchiMDE* possède une vision globale de l'application et peut donc générer ce sérialiseur (voir figure 6.4). Cet exemple montre clairement que les différents profils de développeur peuvent se concentrer sur leur cœur de métier tout en facilitant l'échange et la réutilisation d'informations entre les différents métiers.

Bien que la figure 6.4 représente de façon plutôt séquentielle l'enchainement des différentes transformations, il est possible et fréquent d'itérer sur les premières transformations (fusion *HPCML* et *Paprika.numerical*) sans utiliser les transformations de plus bas niveaux (*HPCML* vers *Arcane* ou *Paprika.numerical* vers *Paprika.hmi*). Cette souplesse dans le développement est en adéquation avec les préconisations de l'approche *MDE4HPC* concernant la mise en place d'un processus de développement adapté au logiciel de calcul scientifique.

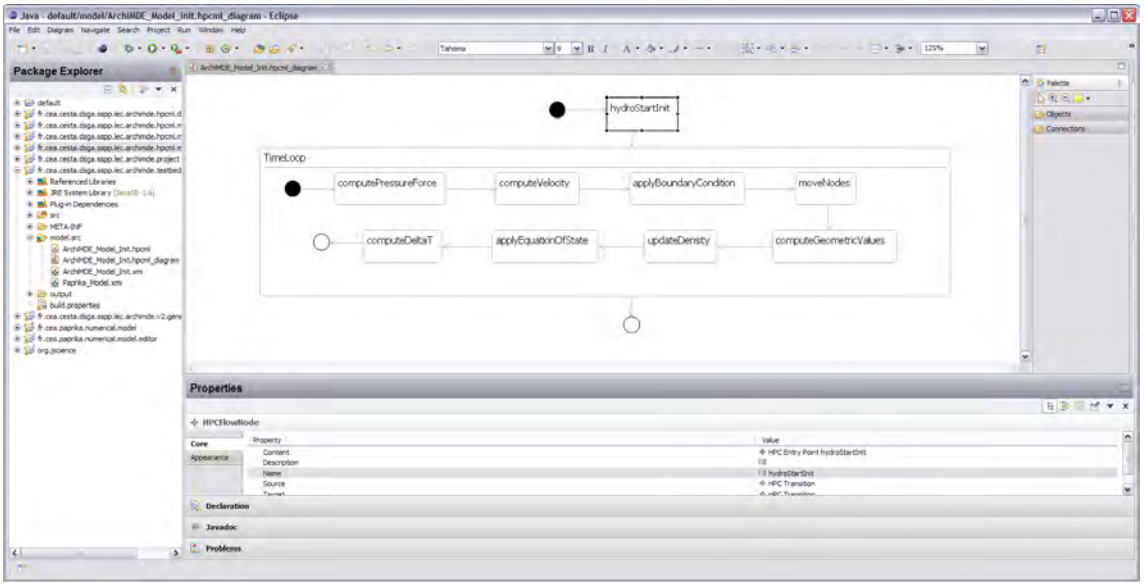


FIGURE 6.5 – Aperçu de l'outil *ArchiMDE*

Un aperçu de l'outil *ArchiMDE* est présenté dans la figure 6.5. On y distingue la perspective HPCML.behavior avec la définition d'un `HPCFlowDescriptor` au centre, la vue propriété en bas à droite et finalement à gauche le navigateur permettant l'exploration des ressources.

6.4 Gestion de l'algorithmique de bas niveau

Nous avons vu dans la section 5.1.4 que la version d'*HPCML* présentée dans cette thèse ne propose pas de formalisme pour la définition d'un `HPCAlgorithmicContent` mais seulement des contraintes sur le formalisme à utiliser. Dans cette section, nous avions aussi abordé le fait que l'outil *ArchiMDE* utilise le langage fourni par la plateforme *Arcane* comme formalisme pour l'algorithmique de bas niveau. Or, il existe plusieurs façons de gérer l'intégration de cette algorithmique avec les modèles de haut niveau. Dans cette section, nous présentons et évaluons quatre modes de gestion que nous avons identifiés. Ce tour d'horizon de la problématique d'intégration des modèles avec l'algorithmique de bas niveau sera l'occasion d'aborder les choix que nous avons effectués pour l'outil *ArchiMDE* ainsi que les perspectives d'évolution envisageables.

6.4.1 Génération directe

Dans cette configuration, le langage de la plateforme cible est utilisé pour décrire les `HPCAlgorithmicContents`. Un attribut de type *String* est utilisé pour stocker le contenu. Lors de la génération, ce contenu est utilisé tel quel par les transformations de modèles. C'est l'approche qui a été choisie dans la version actuelle d'*ArchiMDE* en raison de sa rapidité de mise en œuvre.

ÉVALUATION

+ l'outillage est plus simple à mettre en œuvre et à maintenir que pour les autres approches.
+ le code généré est jetable.
+ les *refactorings* même importants du modèle n'impliquent pas de gestion complexe au niveau du générateur.
+ il est possible de réutiliser des langages existants développés par un tiers.
− il n'est pas possible de modifier le code généré sous peine de perdre la totalité des modifications lors de la prochaine génération.
− il n'y a pas de cohérence entre le code et le modèle. Lors de l'écriture du code il n'y a par exemple ni analyse statique, ni complétion.
− la pérennité de l'algorithmique de bas niveau dépend de celle du langage cible.
− un développement peut être plus complexe pour le développeur dans le cas où la plateforme cible nécessite l'utilisation de plusieurs langages.

6.4.2 Génération incrémentale

Dans cette configuration, le modèle ne contient pas d'information sur l'algorithmique de bas niveau. À partir du modèle, le générateur produit un squelette de l'application dans le langage cible avec des classes et des méthodes abstraites. Le développeur peut alors implémenter l'algorithmique de bas niveau en créant des classes dérivant des classes abstraites. Lorsque le modèle est modifié, le générateur applique les modifications dans le code généré de façon incrémentale. Suite à certains *refactorings*, les classes d'implémentation peuvent devenir non valides vis-à-vis des classes abstraites qu'elles étendent.

ÉVALUATION

+ l'utilisateur développe le code manquant au sein d'un *IDE* dédié au langage cible et bénéficie des fonctionnalités de ce dernier : analyse statique, complétion, etc.
+ il est possible de modifier dans une certaine mesure le code généré via des mécanismes d'extension ou d'annotation.
+ il est possible de réutiliser des langages existants développé par un tiers.
− des *refactorings* importants peuvent poser des problèmes au générateur. Par exemple, lorsqu'on souhaite scinder un composant en deux sous-composants, la propagation de la modification sur l'algorithmique de bas niveau n'est pas triviale.
− la pérennité de l'algorithmique de bas niveau dépend de celle du langage cible.
− la connaissance de l'application est répartie entre les modèles et le code source.
− il est difficile de gérer plusieurs versions d'un même code de simulation.

6.4.3 Algorithmique modèle

Dans cette configuration, l'algorithmique de bas niveau est formalisée par un langage textuel défini grâce à la théorie des modèles plutôt que celle des grammaires. Ce travail nécessite donc la définition d'un métamodèle adapté à, l'algorithmique de bas niveau rencontrée dans le calcul scientifique et, à outiller sa syntaxe concrète textuelle.

Évaluation

+ tout est modèle. Par conséquent, l'intégration et la cohérence entre algorithmique de bas niveau et composants sont assurées.
+ des concepts métiers peuvent être introduits dans l'algorithmique de bas niveau.
+ le code généré est jetable.
+ la pérennité de l'algorithmique de bas niveau est indépendante de celle du langage cible.
+ des transformations de modèles peuvent être appliquées sur l'algorithmique de bas niveau.
− de par sa complexité, car complet au sens de Turing, le développement et la maintenance d'un tel langage aurait un cout élevé.

6.4.4 Synchronisation

Dans cette configuration, l'algorithmique de bas niveau est formalisée par langage dédié embarqué dans un langage hôte généraliste. Dans ce cas, on pourrait définir un nouveau *eDSL* ou utiliser un *eDSL* existant qui possèderait les qualités requises, le langage *Liszt* par exemple (section 3.1.11). Il faudrait, ensuite, synchroniser les modèles de haut niveau et le code écrit avec l'*eDSL*. Le choix d'un langage dynamique comme *Scala* ou *C#* permettrait de faciliter l'implémentation d'un tel système puisqu'ils permettent de récupérer facilement l'arbre syntaxique abstrait d'un code source.

Évaluation

+ la maintenance du langage utilisé est plus facile.
+ le code généré est jetable.
+ la pérennité de l'algorithmique de bas niveau est indépendante de celle du langage cible.
+ des concepts métiers peuvent être introduits dans l'algorithmique de bas niveau
− l'outillage du mécanisme de synchronisation est lourd à mettre en place.

6.4.5 Bilan sur le choix du mode de gestion

Comme nous l'avons évoqué dans la section 4.3, notre objectif était de commencer par la définition et la validation du langage *HPCML*$_n$. Or, celui-ci résulte d'un processus itératif qui a pour conséquence de rendre les modèles et les transformations non conformes à chaque mise à jour du métamodèle. L'activité de définition du langage a donc un impact lourd sur le développement de l'outillage puisque les outils permettant de faciliter la coévolution de modèles n'en sont qu'à leur balbutiement [81, 120]. C'est pour l'ensemble de ces raisons que nous avons choisi l'approche par génération directe.

Lors de la poursuite de la définition de l'approche *MDE4HPC* et notamment du langage *HPCML*$_e$, la génération directe va atteindre ses limites car elle ne permettra pas de réaliser des transformations sur l'algorithmique de bas niveau. Nous pensons que l'approche par synchronisation est, tout en étant pragmatique, capable de répondre aux besoins d'une future version d'*ArchiMDE* qui implémenterait ce langage *HPCML*$_e$.

6.5 Conclusion

Ce chapitre a été l'occasion de présenter l'outil *ArchiMDE*. Nous avons décrit son architecture logicielle et introduit l'*Eclipse Modeling Framework* (*EMF*) qui lui fournit les briques de base. Nous avons détaillé le fonctionnement de l'outil *Paprika* qui utilise les principes de l'*IDM* pour fournir plusieurs types d'éditeurs à partir d'un modèle de jeux de données. Nous avons ensuite présenté comment l'intégration de l'outil *Paprika* au sein d'*ArchiMDE* permet de générer des applications pour la plateforme *Arcane*. Finalement, nous avons abordé le problème de la gestion de l'algorithmique de bas niveau en présentant les solutions que nous avons identifiées et en expliquant pourquoi nous avons choisi la génération directe pour cette version de l'outil *ArchiMDE*.

Bien qu'étant seulement un prototype, cet outil a servi de support à des travaux pratiques lors de l'école d'été internationale organisée par le *CEA*, *EDF* et l'*Inria* en 2010 [1]. Cet événement, dont la thématique était le « *Sustainable High Performance Computing* », a été l'occasion de faire tester l'outil auprès de plus de 40 personnes. Les nombreux retours sur les concepts manquants ou inadaptés ont contribué à faire évoluer l'outil *ArchiMDE* ainsi que le langage *HPCML*$_n$ (section 4.2.2).

1. http://www-hpc.cea.fr/fr/actualites2010.htm

Chapitre

7

Étude de cas : code d'électromagnétisme 3D

Il n'y a pas de création sans épreuve...
Fernand Ouellette.

Sommaire

L'approche *MDE4HPC* a fait l'objet d'une évaluation en deux étapes. Dans un premier temps, nous avons utilisé l'outil *ArchiMDE* présenté dans le chapitre précédent pour développer le code simplifié d'hydrodynamique lagrangienne proposé dans [106]. L'évaluation de ce développement, dont les résultats ont été publiés [168], nous a permis de valider le fait qu'il est possible de générer avec l'approche *MDE4HPC* du code viable en termes de performance et surtout qu'il est possible d'obtenir une réduction des couts de maintenance. Dans un second temps nous souhaitions évaluer la capacité du langage $HPCML_n$ à modéliser une application utilisée en production et basée sur un autre domaine physique.

Ce chapitre aborde le développement d'un code de simulation permettant de calculer la diffraction d'une onde électromagnétique plane par un objet 3D. Nous commencerons dans la section 7.1, par brièvement présenter le problème modélisé par le code de simulation ainsi que la méthode de résolution employée. Ensuite, dans la section 7.2, nous aurons l'occasion de présenter plusieurs modèles $HPCML_n$ issus de ce développement.

7.1 Présentation du problème simulé

Le code de simulation modélisé dans ce chapitre est basé sur un code utilisé en production qui permet de simuler la diffraction d'une onde électromagnétique plane par des objets 3D complexes composés de plusieurs matériaux pouvant être conducteurs ou diélectriques. Dans notre cas, nous nous sommes concentrés sur les objets 3D mono-matériaux de type conducteur.

Problème Un corps conducteur parfait est éclairé par une onde incidente plane harmonique. La présence de cet obstacle modifie les champs incidents : c'est ce qu'on appelle le phénomène de diffraction. Notre but est de calculer les courants électriques induits par cette onde incidente sur la surface de l'objet.

Modèle physique Le modèle physique choisi pour l'étude de ce phénomène repose sur les équations de Maxwell [118] :

$$\nabla \times \mathbf{E}(r,t) = -\frac{\partial \mathbf{B}(r,t)}{\partial t}$$

$$\nabla \times \mathbf{H}(r,t) = \frac{\partial \mathbf{B}(r,t)}{\partial t} + \mathbf{J}_s(r,t)$$

$$\nabla \cdot \mathbf{D}(r,t) = \rho_e$$

$$\nabla \cdot \mathbf{B}(r,t) = 0$$

où l'on note :

$\mathbf{E}(r,t)$: champ électrique $(V.m^{-1})$
$\mathbf{H}(r,t)$: champ magnétique $(A.m^{-1})$
$\mathbf{D}(r,t)$: champ d'induction électrique $(C.m^{-2})$
$\mathbf{B}(r,t)$: champ d'induction magnétique (T)
$\mathbf{J}_s(r,t)$: densité de courant électrique source $(A.m^{-2})$
ρ_e : densité volumique de charges électriques $(C.m^{-3})$

Modèle mathématiques de résolution Nous allons nous baser sur une formulation en équations intégrales surfaciques des équations de Maxwell en régime harmonique pour la résolution. On ramène ainsi un problème tridimensionnel posé sur un domaine non borné à un problème borné posé sur une surface.

Les équations intégrales sous forme variationnelle que l'on se propose de résoudre sont les équations classiques de la *EFIE* (Electric-Field Integral Equation) [59] :

$$\iint_{\Gamma\times\Gamma} \phi(y-x)[i\omega\mu J(x)J'(x) + \left(\frac{1}{i\omega\varepsilon}\right) \nabla(div_\Gamma J(x))]div_\Gamma J'(y)d\gamma(x)d\gamma(y) = -\int_\Gamma E^{inc}(x).J'(x)d\gamma(x)$$

où l'on note :

J : l'inconnue
J' : une fonction test
μ : permittivité du milieu $= \left(\frac{1}{36}\pi\right) \times 10^{-9}\ F.m^{-1}$ car nous travaillons dans le vide
ε : perméabilité magnétique $= 4\pi \times 10^{-7}\ H.m^{-1}$ car nous travaillons dans le vide
f : fréquence choisie (Hz)
c : vitesse de la lumière $= 3 \times 10^8\ m.s^{-1}$
ω : $2\pi \left(\frac{f}{c}\right)$
ϕ : noyau de Green

Nous utilisons ensuite la méthode des éléments finis pour résoudre ces équations intégrales. Les principales étapes de construction d'un modèle éléments finis sont les suivantes [166] :

- discrétisation du milieu continu en sous domaines (le maillage) ;
- construction de l'approximation par sous domaine (choix des éléments finis) ;
- calcul des matrices élémentaires correspondant à la formulation variationnelle du problème ;
- assemblage des matrices élémentaires ;
- prise en compte des conditions aux limites ;
- résolution du système d'équations linéaires de type $\mathbf{Ax} = \mathbf{b}$.

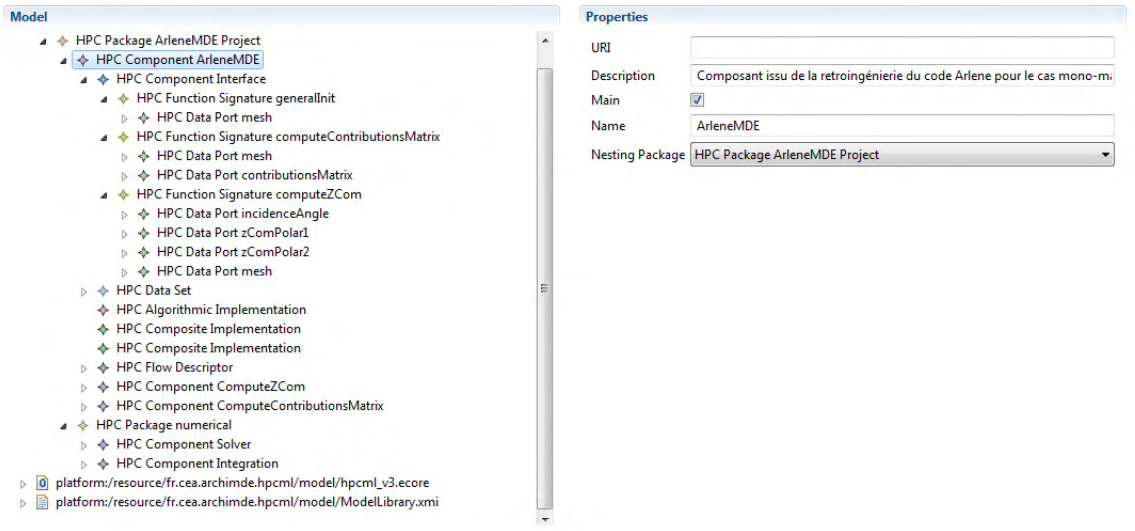


FIGURE 7.1 – Vue générale de la partie structurelle

7.2 Modélisation de l'application avec HPCML

Cette section a pour objectif de montrer concrètement à quoi ressemble la modélisation, selon l'approche *MDE4HPC*, d'une application de simulation numérique. Après quelques remarques générales sur le processus de modélisation, nous aborderons des exemples de modèles concernant la résolution du problème que nous venons d'exposer.

7.2.1 Processus de modélisation

Le code existant, dont sont issus les modèles présentés dans cette section, est écrit en *Fortran 90* et utilise le standard *MPI* pour la gestion du parallélisme. La montée en abstraction effectuée au moment de la (rétro)modélisation n'a pu être réalisée qu'avec la participation des numériciens qui ont transmis leur savoir lors de discussions plus ou moins formalisées. Même si la démarche effectuée (rétroconception) dans ce cas d'étude n'est pas identique à celle d'un nouveau développement, nous avons pu identifier un processus général qui semble bien adapté aux deux cas.

La première étape consiste, lors des discussions sur le modèle mathématique, à dessiner sur une feuille de papier, la hiérarchie et l'enchainement des fonctions en utilisant une syntaxe concrète simplifiée d'*HPCML*. Une fois cette première ébauche terminée, on détermine quelles sont les fonctions disponibles dans les composants existants. On peut ensuite modéliser l'application en suivant le fil conducteur que l'on vient d'établir (enchainement des fonctions et leurs origines). Finalement, on procède à un *refactoring* global de l'application pour déterminer si on a identifié toutes les sources potentielles de parallélisme.

7.2.2 Aperçu général

Cette section donne un aperçu général de l'application modélisée au travers de la présentation de plusieurs modèles. La figure 7.1 permet de découvrir l'organisation à haut niveau des composants. On trouve deux paquetages (`HPCPackage`) principaux : le premier contenant les composants propres au code de simulation que nous développons (*ArleneMDE Project*) et le deuxième contenant des composants fournissant des services numériques génériques (résolution de système linéaire et intégration).

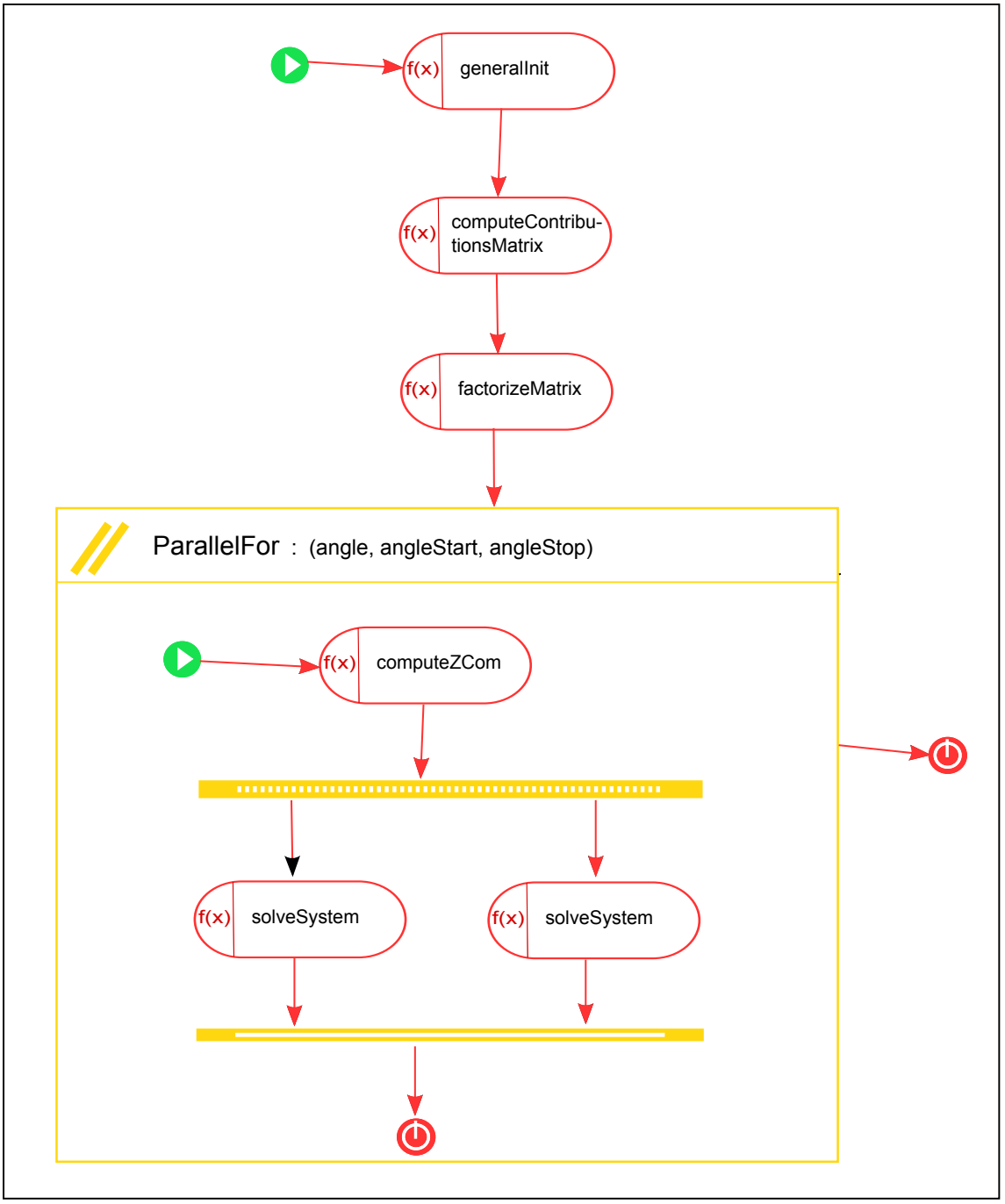


FIGURE 7.2 – `HPCFlowDescriptor` du composant *ArleneMDE* avec seulement l'affichage du flot d'exécution

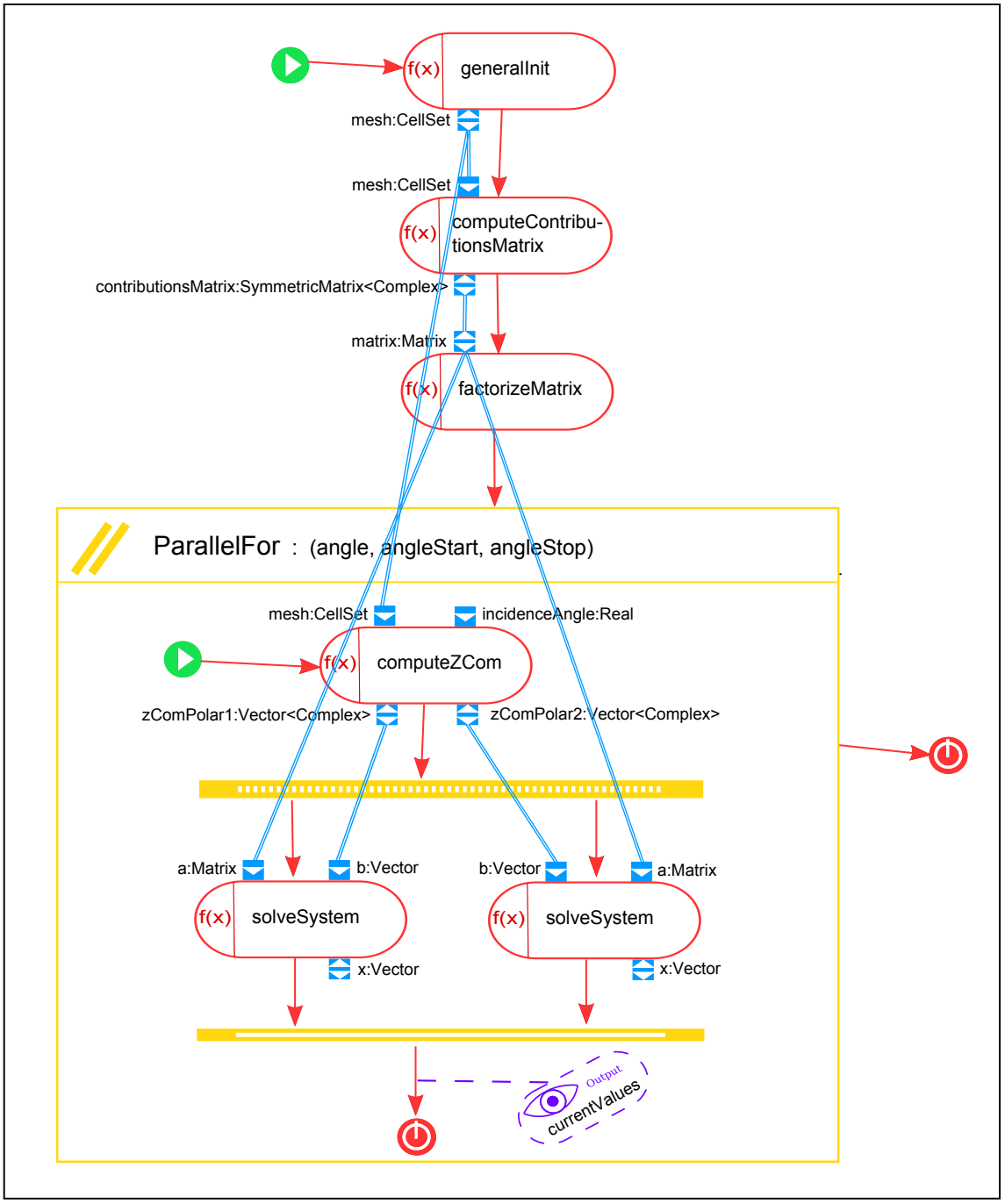


FIGURE 7.3 – `HPCFlowDescriptor` du composant *ArleneMDE* avec l'affichage des données et des sorties activés

On aperçoit aussi sur la figure 7.1 une partie de la structure du composant principal de l'application (`HPCComponent` *ArleneMDE* avec la propriété `main` à *true*) et notamment la définition de son interface (`HPCComponentInterface`). Il est intéressant de noter que même au premier niveau de hiérarchie du composant d'application *ArleneMDE*, on trouve les deux sortes d'implémentations (`HPCCompositeImplementation` et `HPCAlgorithmicContent`).

Si on fait le lien avec la section 5.2.3 qui définit la syntaxe concrète de la partie structurelle, on retrouve bien un arbre représentant la hiérarchie des éléments du modèle qui s'insère dans une vue à deux zones de type *master/details*.

On découvre dans la figure 7.2, l'`HPCFlowDescriptor` du composant *ArleneMDE* où seulement le flot d'exécution est visible. Ce `HPCFlowDescriptor` utilise plusieurs formes d'expression du parallélisme : un `HPCParallelFor` et le couple `HPCFork`/`HPCJoin`. Notons que l'imbrication de plusieurs constructions pour exprimer le parallélisme permet de les combiner pour obtenir encore plus de parallélisme.

La figure 7.3 montre le même `HPCFlowDescriptor` mais avec des paramètres d'affichage différents puisque les flots de données et les aspects concernant les sorties sont affichés. Nous illustrons ici la notion de filtre d'affichage tel que nous l'avons définie dans la section 5.2.2. On remarque immédiatement que le diagramme perd en lisibilité, ce qui justifie la nécessité des filtres. Toutefois, la vocation principale de ce mode d'affichage est de faciliter la connexion des ports de données (`HPCDataPort`) entre eux.

7.2.3 Calcul de la matrice des contributions

Cette section est consacrée au composant *ComputeContributionsMatrix* qui est un sous-composant du composant *ArleneMDE* et qui est utilisé pour raffiner la spécification du comportement de ce dernier.

La figure 7.4 permet de découvrir l'organisation de ce composant.

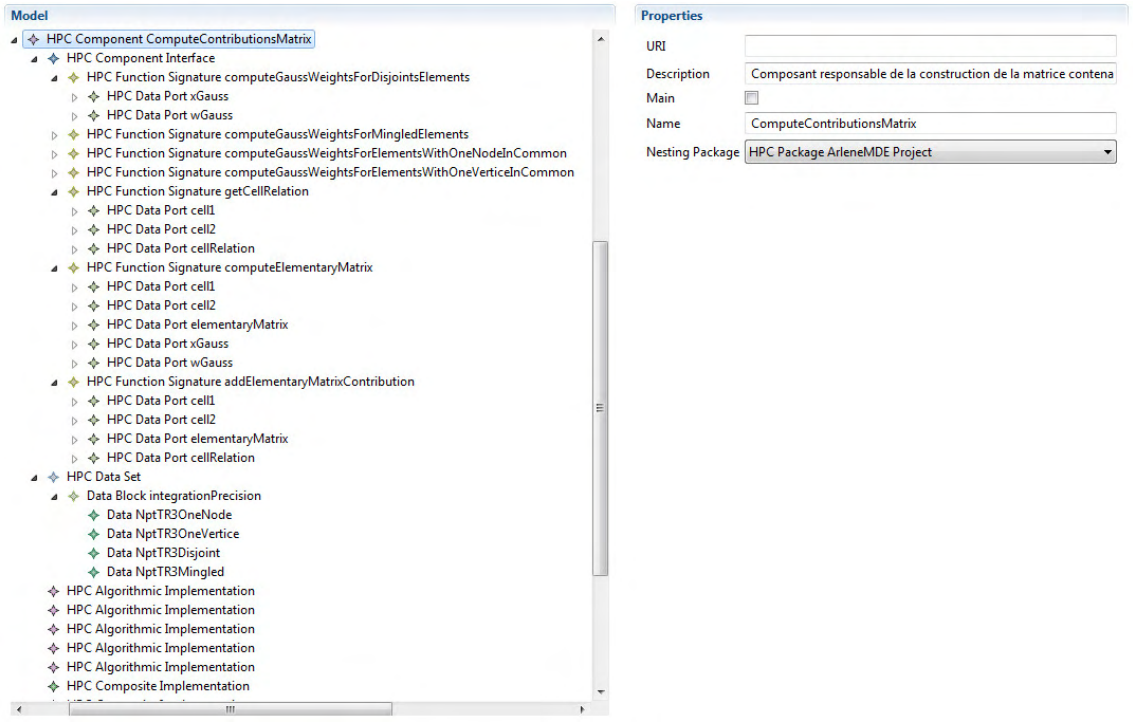


FIGURE 7.4 – Aperçu de la partie structurelle du composant *ComputeContributionsMatrix*

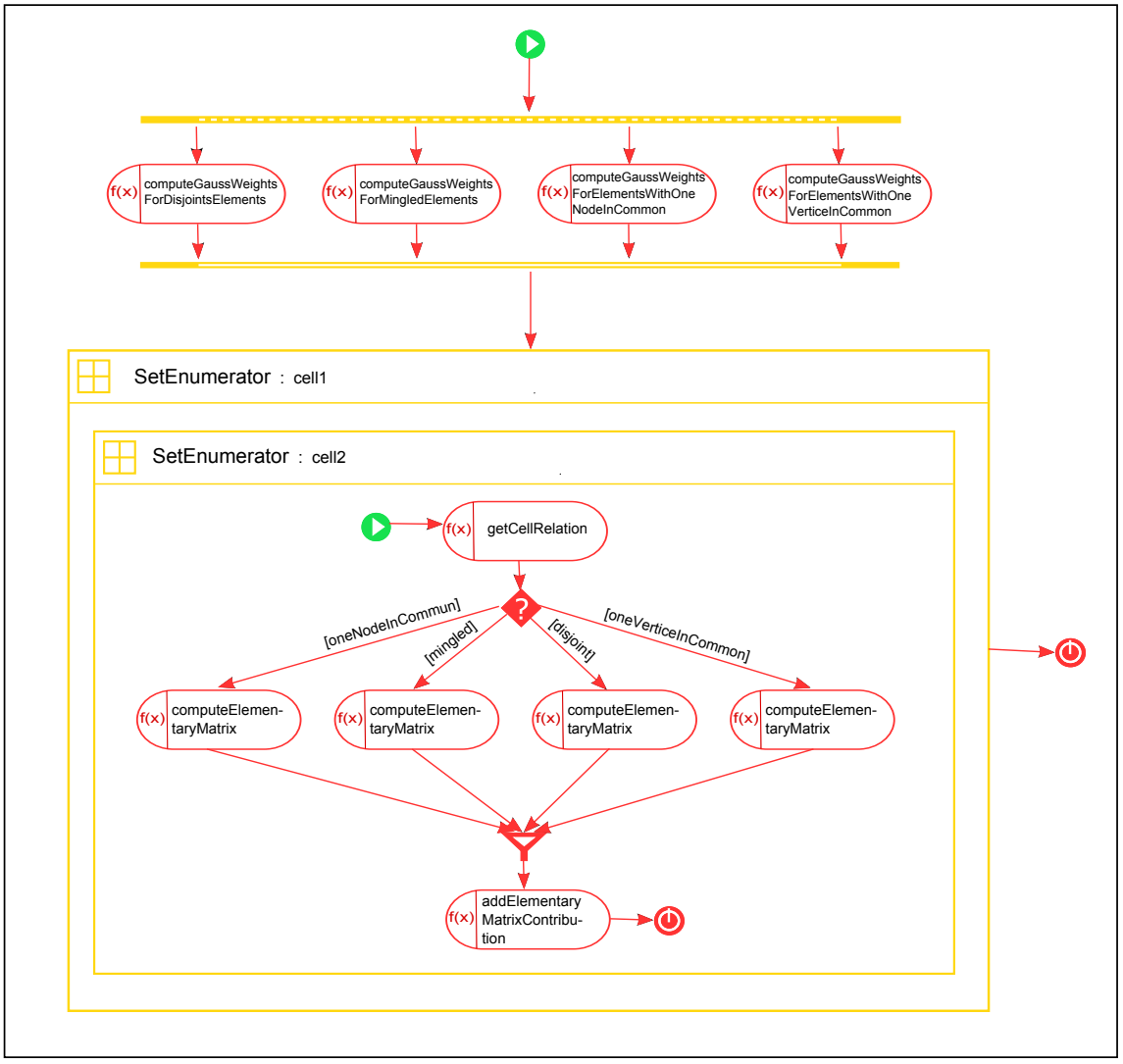


FIGURE 7.5 – `HPCFlowDescriptor` du composant *ComputeContributionsMatrix*

La figure 7.5, quant à elle, montre l'`HPCFlowDescriptor` du composant avec le flot de contrôle. On distingue quelques éléments nouveaux par rapport à l'`HPCFlowDescriptor` du composant *ArleneMDE* présenté dans la figure 7.3.

Premièrement, le composant *ComputeContributionsMatrix* est un sous-composant du composant *ArleneMDE*, par conséquent son `HPCFlowDescriptor` possède une `HPCFunction-Signature` qui découle de l'une des `HPCFunctionSignatures` de l'interface du composant *ArleneMDE* : il s'agit de la signature *computeContributionsMatrix* dont la déclaration est visible dans la figure 7.1. C'est pour cette raison qu'on note la présence de ports de données (`HPCDataPort`) sur les bords du diagramme, ce sont ceux définis par l'interface.

Deuxièmement, on découvre une mise en application de la troisième façon d'exprimer du parallélisme, l'`HPCSetEnumerator`. On retrouve même ici l'imbrication de deux `HPCSetEnumerators` puisque l'on souhaite calculer la contribution de chaque maille envers chaque maille. Notons que la valeur du `stencil` de ces deux `HPCSetEnumerators` est *all*, puisque chaque maille va devoir accéder à toutes les autres mailles du moins en lecture (`HPCDataPort` de la relation `set` en mode `IN`). C'est typiquement un cas qui va poser plus de problème lors de la génération de l'implémentation vers des architectures à mémoire

distribuée que sur celles à mémoire partagée. Dans le cas de la mémoire partagée, on peut imaginer que toutes les unités d'exécution accèdent aux données du maillage par leur mémoire commune. Dans le second cas, on peut imaginer plusieurs solutions dont l'une d'entre elles serait de répliquer les données du maillage sur la mémoire de chaque unité d'exécution. Cette solution n'est bien sûr envisageable que si la mémoire des unités d'exécution est suffisamment grande pour recevoir les données du maillage.

Finalement, la figure 7.5 contient la dernière structure de flot contrôle qui n'avait pas été présentée : les branchements conditionnels (`HPCIf` et `HPCEndIf`).

7.3 Conclusion

Dans ce chapitre, nous avons présenté un cas d'étude qui nous a permis d'appréhender le comportement du langage *HPCML* lors de la modélisation d'un domaine physique différent du premier cas d'étude [168] que nous avons présenté dans l'introduction de ce chapitre. Ce premier cas d'étude avait montré la viabilité en termes de performance du code généré par l'approche *MDE4HPC* ainsi qu'une réduction des couts de maintenance.

Le modèle de l'application présentée dans ce chapitre utilise tous les concepts que nous avons proposé pour exprimer le parallélisme. Ce modèle est donc une base adéquate pour conduire les expérimentations nécessaires à la définition de la couche $HPCML_e$ dont nous allons parler dans le chapitre 9.

Chapitre

8

Atteinte des critères d'évaluation

On est souvent injuste par omission.

Marc Aurèle.

Sommaire

Ce chapitre a pour objectif de positionner l'approche *MDE4HPC* vis-à-vis des critères d'évaluation que nous avons définis pour réaliser notre état de l'art, à savoir : portabilité, abstraction et accessibilité, séparation des préoccupations, communauté et outillage, validation (se référer à l'annexe A pour plus de détails).

De nombreuses comparaisons de l'approche *MDE4HPC* seront faites par rapport à des développements réalisés de façon « classique ». Par le terme « développement classique », nous faisons référence aux développements logiciels considérant le code source comme un élément de premier plan et où les modèles, s'ils sont présents, ne jouent qu'un rôle purement « contemplatif ».

8.1 Portabilité

Afin d'évaluer la portabilité des logiciels de simulation numérique développés avec l'approche *MDE4HPC*, nous allons prendre deux cas d'utilisations faisant intervenir cette caractéristique : le développement d'une application et la mise en production d'une nouvelle machine de calcul tel un supercalculateur.

Pour cette évaluation, nous prendrons comme hypothèse que la gestion de l'algorithmique de bas niveau est implémentée sous une forme permettant la génération multiple facilement : algorithmique modèle ou synchronisation (section 6.4).

8.1.1 Cas d'un nouveau code de simulation

Pour cette situation, nous considérons que les générateurs intégrant les différentes transformations de modèles permettant de passer des modèles abstraits au logiciel de simulation sont disponibles. Leurs couts de développement seront pris en considération dans le prochain cas (mise en place d'un nouvelle machine).

En comparaison d'un « développement classique » où il faudrait développer une version pour chaque architecture cible, l'approche *MDE4HPC* ne nécessite qu'un seul développement. À noter qu'en fonction de l'écart existant entre les architectures matérielles, les différences entre plusieurs versions de l'application peuvent être plus ou moins grandes. Dans le meilleur cas, c'est le compilateur ou les couches logicielles basses qui vont prendre en charge les différences. Par contre, si les architectures sont radicalement différentes, la durée de la phase de conception et de la phase de codage seront alors fortement impactées puisqu'il y a peu d'éléments réutilisables entre les deux développements.

Un autre aspect à prendre en compte concerne les couts de formation sur les solutions logicielles permettant d'exploiter la machine cible. Dans un « développement classique », tous les développeurs doivent être formés aux spécificités et aux solutions logicielles des machines cibles. Sachant que ces développeurs ont le plus souvent une formation en mathématiques appliquées, la charge d'apprentissage est lourde. *MDE4HPC* améliore la situation en proposant de mutualiser ce savoir pour tous les projets au travers du développement des transformations de modèles embarquées dans les générateurs.

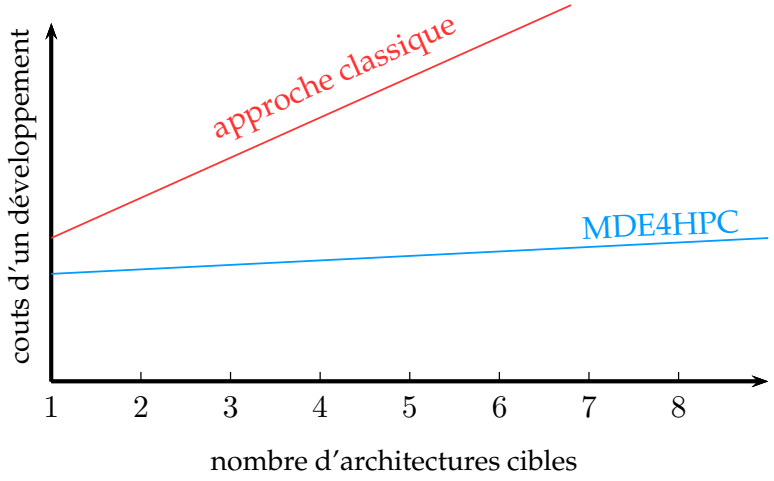


FIGURE 8.1 – Comparaison des couts de développement d'un nouveau logiciel entre une approche classique et *MDE4HPC*

Si l'on tente d'exprimer les composantes des couts des deux approches de développement on obtient les formules suivantes [1] :

$$C_{devclassique} \leq C_{specification} + \sum_{i=1}^{na}(C_{conception_i} + C_{codage_i} + C_{formation_i} + C_{test_i})$$

$$C_{devMDE4HPC} = C_{specification} + C_{specification_{detaillee}} + \sum_{i=1}^{na}(C_{parametrage-generateur_i})$$

La formule du « développement classique » concerne le pire cas, c'est-à-dire celui où il n'existerait pas de solution logicielle capable d'exploiter plusieurs des architectures cibles. En réalité, il est parfois possible de réutiliser une partie de la conception et du codage entre plusieurs développements.Pour l'approche *MDE4HPC*, le cout des tests n'est pas explicité dans la formule car on considère qu'ils sont générés à partir de la spécification. Bien que

1. C = cout, na = nombre d'architectures cibles

ne disposant pas d'assez de données empiriques pour quantifier finement les couts des composantes des deux approches, on peut établir l'allure générale des courbes associées à ces équations. Elles sont présentées dans la figure 8.1.

On rappelle que dans le scénario présenté ici, on a fait l'hypothèse que les générateurs sont disponibles. Globalement, le seul constat que l'on puisse établir est que le cout d'un « développement classique » évolue linéairement avec le nombre d'architectures cibles alors qu'avec l'approche *MDE4HPC* on se rapproche d'un cout constant. En effet, on peut considérer le paramétrage du générateur comme étant une tâche négligeable vis-à-vis de l'ensemble des activités nécessaires dans le cas d'un développement classique.

8.1.2 Cas d'un changement de machine

La mise en production d'une nouvelle machine de calcul signifie que l'ensemble des codes de simulation doit être porté sur cette dernière. Le terme de *portage* fait ici référence à trois sous activités dans le cadre d'une « approche classique » : l'adaptation ou la traduction du code existant afin qu'il puisse exploiter la nouvelle architecture, l'optimisation du code en utilisant les spécificités de la nouvelle architecture et finalement la validation de la nouvelle version pour s'assurer qu'il n'y pas eu de régressions fonctionnelles. Ces activités sont à réaliser sur l'ensemble des codes. Cependant, en fonction des solutions logicielles choisies, ces activités seront plus ou moins lourdes à réaliser. Par exemple, la lourdeur des migrations de codes de simulation reposant sur un *framework* tel qu'*Arcane* (section 3.1.12) peut être grandement allégée lorsqu'il est possible de prendre en compte le changement d'architecture au niveau du *framework*.

Avec l'approche *MDE4HPC*, il est nécessaire de développer un nouveau générateur et de l'appliquer sur l'ensemble des modèles d'applications en le paramétrant. En fonction de la radicalité des changements d'architectures matérielles et logicielles on pourra espérer réutiliser des éléments, notamment des transformations de modèle, provenant d'autres générateurs.

Dans ce nouveau cas d'utilisation, lorsqu'on tente d'exprimer les composantes des couts des deux approches de développement, on obtient les formules suivantes [2] :

$$C_{portage_{classique}} \leq \sum_{i=1}^{nc}(C_{adaption_i} + C_{optimisation_i} + C_{formation_i} + C_{test-fonctionnel_i})$$

$$C_{portage_{MDE4HPC}} = C_{generateur} + \sum_{i=1}^{nc}(C_{utilisation-generateur_i})$$

Comme pour la section précédente, la formule du cout de portage pour un « développement classique » représente le pire cas. Cependant, ce terme majorant n'est pas très loin de la réalité, car le seul cout qui peut être mutualisé entre plusieurs développements est celui de la formation pour la maitrise des nouvelles solutions logicielles. Or, il est impossible de sortir ce terme de la somme car comme les codes sont, en général, développés par des équipes différentes, chacune de ces équipes doit se former à ces nouvelles solutions logicielles.

Pour l'approche *MDE4HPC*, le cout des tests fonctionnels ne sont pas explicités car comme dit dans la section précédente, ils sont générés à partir de la spécification. Ils ont néanmoins un impact sur le développement du générateur puisque ce dernier doit être en mesure de générer les tests pour la nouvelle architecture.

L'allure générale des courbes associées à ces équations est présentée dans la figure 8.2 :

2. C = cout, nc = nombre de code de simulation à migrer

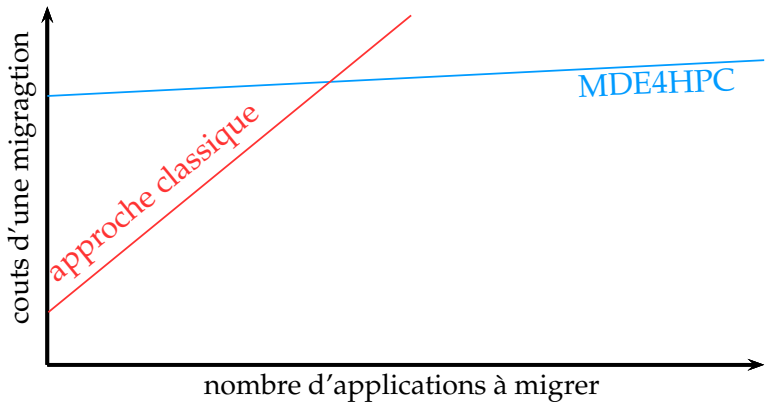


FIGURE 8.2 – Comparaison des couts de migration vers une nouvelle architecture entre une approche classique et *MDE4HPC*

Le modèle de cout de portage dans le cadre d'une « approche classique » est très proche d'un comportement linéaire fonction du nombre d'applications à migrer. De son coté, l'approche *MDE4HPC* possède un cout fixe initial important, celui du générateur. Ce cout initial est typique de l'*IDM*. Fleurey et coll. font, par exemple, mention du même comportement dans [87] qui relate l'utilisation de l'*IDM* pour la migration d'applications bancaires. Une fois le générateur développé, l'intervention manuelle est beaucoup moins présente que dans une approche classique. Cette différence explique pourquoi le coefficient de la droite *MDE4HPC* sur le graphique est beaucoup plus faible que celui de l'approche classique.

On peut affirmer qu'il existe d'après ces deux formules, un nombre n d'applications à partir duquel l'approche *MDE4HPC* serait moins couteuse qu'une « approche classique » pour le cas d'utilisation d'une migration. Par contre, et à la vue des données que nous possédons, il n'est pas possible d'établir une valeur précise de n. On peut toutefois estimer que l'approche est rentable pour un nombre d'applications inférieur à une dizaine ($2 \leq n \leq 10$). On peut aussi faire l'hypothèse que ce nombre ne varie pas en fonction de l'importance de la migration à réaliser. Par exemple, si les différences entre les deux architectures matérielles sont faibles, la migration en suivant une approche classique sera rapide mais il en sera de même pour le développement du générateur si l'on choisit l'approche *MDE4HPC*.

8.1.3 Remarques générales sur la portabilité

Nous venons d'aborder les deux cas d'utilisations principaux faisant intervenir la notion de portabilité. Les transformations de modèles, éléments de base des générateurs, sont à la fois une force et une faiblesse de l'approche *MDE4HPC* dans le cadre de la portabilité. D'un coté, elles sont un élément clé permettant de traduire les concepts abstraits vers n'importe quelle plateforme cible (notion d'indépendance) tout en mutualisant la connaissance sur ces dernières (les optimisations par exemple). D'un autre coté, elle induisent un investissement initial lourd dans le cadre du développement du générateur qui les utilise.

Afin de réduire le cout de développement d'un générateur pour un organisme comme le *CEA*, on peut imaginer reporter le cout du développement des transformations sur le constructeur de la machine. De la même façon que les constructeurs livrent des versions surmesure de certaines bibliothèques comme *MPI* pour leur machines. On peut ainsi imaginer demander à un constructeur de livrer les transformations de modèle permettant de générer du code optimisé pour les machines qu'il livre. Même si ces transformations pourraient faire

l'objet de personnalisation par le client, la plus grosse partie du générateur serait disponible. De surcroit, cette approche permettrait au constructeur de transmettre son savoir métier au client en formalisant dans les transformations sa connaissance de la machine.

8.2 Abstraction et accessibilité

Les langages dédiés permettent d'exprimer des solutions dans un idiome proche du domaine métier concerné avec, qui plus est, le juste niveau d'abstraction. Comme le montre Van Deursen et coll. en se basant sur leur bibliographie au sujet des *DSL* [208], ils permettent directement aux experts métiers de comprendre, valider, modifier et souvent développer des programmes basés sur ces *DSL*.

La question est donc de savoir si nous pouvons qualifier le langage *HPCML* de langage dédié. Il n'existe pas de définition précise et reconnue donnant les caractéristiques d'un *DSL*. D'ailleurs, peut-on vraiment affirmer qu'un langage est spécialisé pour un domaine ou ne l'est pas ? Certains considèrent le langage *COBOL* (*COmmon Business Oriented Language*)[165] comme un langage dédié alors que d'autres estiment que cela outrepasse la notion de domaine. Face à ces divergences d'opinions, est-ce qu'il n'est pas plus naturel de considérer divers degrés pour cette propriété comme le proposent par exemple Mernik et coll. [153] ? On obtiendrait alors non pas d'un coté les *DSL* et de l'autre les langages généralistes, mais une échelle où seraient placés les langages en fonction de leur degré de spécialisation dans un domaine.

Dans ces conditions, il faut identifier quels sont les éléments du langage *HPCML* qui témoignent de sa spécialisation dans le domaine de la simulation numérique haute-performance. En analysant les éléments présentés dans le chapitre 5, on identifie notamment :

- *les types de données* : le langage *HPCML* possède des concepts permettant de définir les types de données manipulés dans une simulation numérique (section 5.1.2) : types numériques simples comme les entiers, les réels ou les complexes (`HPCNumericalType`) ; des types composés comme les coordonnées ou les tenseurs (`HPCStructureType`) ; des types basés sur des tableaux comme les vecteurs ou les matrices (`HPCArrayType`).
  Dans le cas des simulations numériques basées sur la résolution des *EDP*, Berti [33] a identifié la nécessité de fournir des types représentant les éléments d'un maillage (nœud, arête, face, maille) ainsi que la possibilité de définir des ensembles de ces éléments. *HPCML* offre cette possibilité au travers des concepts `HPCMeshElementType` et `HPCSetType`. Berti indique aussi la nécessité de pouvoir associer différents types de données à des éléments de maillage. Le langage *HPCML* remplit cette condition grâce aux concepts `HPCMesh` et `HPCField`.
- *la gestion des constantes* : le langage *HPCML* gère les constantes d'une manière bien particulière. Les constantes, et notamment les constantes physiques, possèdent un rôle très important dans une simulation numérique. C'est pourquoi *HPCML* propose une approche permettant de faciliter leur cohérence au sein d'un code de simulation.
- *la gestion du parallélisme* : le langage *HPCML* fournit des concepts abstraits qui représentent des stratégies de parallélisme. Parmi ces concepts, on trouve l'`HPCSetEnumerator`, dont le parallélisme repose sur la décomposition du domaine de calcul, c'est-à-dire le maillage. C'est donc un concept propre à la simulation numérique, et plus particulièrement, à la résolution d'*EDP*.
- *les études paramétriques* qui peuvent être définies via l'ensemble des concepts du paquetage *parametric* (section 5.1.7).

L'ensemble de ces éléments nous permet de conclure que le langage *HPCML* se situe plutôt du coté des langages dédiés que de celui des langages généralistes et que, par consé-

quent, on peut lui attribuer les qualités reconnues des premiers cités dont font parties l'abstraction et l'accessibilité.

8.3 Séparation des préoccupations

Une des vocations de l'ingénierie dirigée par les modèles (section 4.1) est de faciliter la séparation des préoccupations au travers de la composition de modèles. L'approche *MDE4HPC* exploite cette possibilité en offrant différentes perspectives de modélisation situées à plusieurs niveaux d'abstraction (voir chapitre 4 et section 6.3).

La séparation des préoccupations entre la spécification de la méthode de résolution mathématique et l'expression du parallélisme est certainement le point qui fait le plus défaut dans les solutions logicielles existantes (section 3.1.14). Pour répondre à ce problème, le langage *HPCML* propose des concepts qui encapsulent des stratégies de parallélisme (voir section 5.1.4). Ces concepts ont pour objectif d'altérer au minimum la description du schéma numérique tout en permettant de spécifier les sources potentielles de parallélisme.

Le langage *HPCML* propose aussi de définir certaines préoccupations sous formes d'aspects. Ces aspects, qui peuvent concerner la validation (section 5.1.6) ou la spécification des sorties de la simulation (section 5.1.5), sont définis en dehors des flots d'exécution et de données. Il sont rattachés à l'application en indiquant leurs points d'insertion.

8.4 Validation

Le langage *HPCML* offre plusieurs mécanismes pour améliorer la robustesse des logiciels et pour faciliter les activités de validation. Premièrement, de par sa construction et ses contraintes, le langage diminue les sources d'erreurs en restreignant le nombre de constructions valides. Les vérifications réalisées de manière statique sont donc déjà très importantes.

Deuxièmement, *HPCML* possède des concepts intrinsèques servant à la validation d'un code de simulation. Ces concepts, présentés dans la section 5.1.6, permettent d'une part de faire de la conception par contrat au travers de la définition de *pré-* et *post-conditions* ; et permettent d'autre part, de définir des aspects orientés validation. Ces deux approches autorisent la génération de tests à des fins diverses : validité d'un assemblage de composants, validité fonctionnelle, etc.

8.5 Communauté et outillage

Comme nous l'avons vu dans la section 6.1, l'outil *ArchiMDE* est basé sur la plateforme *Eclipse*. Or, la plateforme possède une large communauté comprenant plus de 200 projets *open source*, environ un millier de développeurs, plus de 170 sociétés partenaires, des milliers de sociétés basant leurs produits sur la plateforme et, finalement plusieurs millions d'utilisateurs [7]. L'outil *ArchiMDE* hérite de la diversité et de la pérennité de tous ces projets. Même si l'on prend comme hypothèse l'arrêt du développement de la plateforme *Eclipse*, les modèles pourront être facilement migrer vers une nouvelle plateforme puisqu'il sont stockés dans un format standard : le format *XMI*.

L'approche *MDE4HPC* propose la mise en place de processus de générations produisant des applications basées sur des solutions logicielles existantes. Par ce biais, on bénéficie de l'écosystème logiciel (débogueur, compilateur) de ces solutions cibles.

Du point de vue de la communauté utilisateur, l'outil *ArchiMDE* n'est qu'au stade de prototype de recherche et ne possède par conséquent qu'un faible nombre d'utilisateurs.

8.6 Évaluation globale

Dans ce chapitre nous avons évalué l'approche *MDE4HPC* par rapport au critères que nous avons définis pour conduire notre état de l'art. Cependant, l'approche *MDE4HPC* apporte des améliorations en dehors de ces critères. Par exemple, le langage *HPCML* permet notamment de formaliser la spécification des codes de simulations. Or, cette caractéristique est un facteur déterminant dans la capitalisation et la transmission du savoir métier.

Pour conclure ce chapitre, la figure 8.3 montre l'évaluation synthétique de l'approche *MDE4HPC*. On notera, sans surprise, qu'elle répond avantageusement à notre problématique. Seule ombre au tableau, l'atteinte du critère « communauté et outils » qui se situe au niveau 1 à cause de l'outil *ArchiMDE* comme il n'est qu'au stade de prototype de recherche et à cause du faible nombre d'applications développés avec celui-ci.

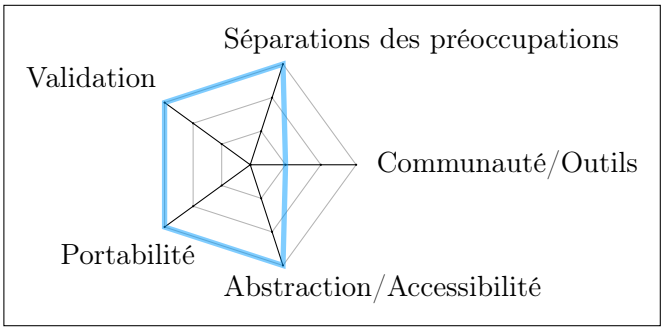


FIGURE 8.3 – Évaluation synthétique de l'*MDE4HPC*

Quatrième partie

Perspectives et conclusion

Chapitre

9

Vers une approche globale basée sur la modélisation

La science progresse en indiquant l'immensité de l'ignoré.
Louis Pauwels.

Sommaire

Un des objectifs de cette thèse est d'identifier les possibilités offertes par l'*IDM* dans le cadre du développement d'applications de calcul scientifique. C'est pourquoi ce chapitre se trouve à cheval entre la partie contribution et la partie perspectives.

Nous avons expliqué dans la conclusion du chapitre présentant l'approche *MDE4HPC* (section 4.3) les raisons qui nous ont amenées à nous concentrer sur le langage $HPCML_n$ (*numerical*). Cependant les langages $HPCML_p$ (*physics* présenté dans la section 4.2.2) et $HPCML_e$ (*execution* présenté dans la section 4.2.2) ont fait l'objet de nombreuses réflexions.

Ce chapitre a donc pour objectif de présenter à la fois les pistes de recherches que nous avons identifiées pour le langage $HPCML_n$ présenté dans le chapitre 5 et celles concernant la définition des langages $HPCML_p$ et $HPCML_e$.

9.1 Perspectives pour le langage HPCMLp

Les problèmes adressés par cette partie du langage *HPCML* ne sont généralement pas considérés comme une priorité de la communauté par rapport à la performance. Cependant, il existe des perspectives intéressantes en termes de capitalisation de la connaissance et de la traçabilité des exigences. Il faudrait pour cela modéliser le modèle physique et surtout capturer la genèse de ce dernier. Des travaux, comme la recommandation *Mathematical Markup Language* (*MathML*) du W3C [211], sont une bonne base pour la définition d'un tel langage.

Il existe des perspectives de recherches intéressantes en ce qui concerne la gestion des incertitudes. La modélisation de ces incertitudes permettrait, avec l'aide d'outils mathématiques appropriés, de déterminer le domaine de validité d'un code de simulation numérique.

9.2 Perspectives pour le langage HPCMLn

Cette thèse s'est focalisée sur la définition du langage *HPCML$_n$*. Cependant, des perspectives importantes d'évolution existent. Nous abordons dans cette section deux d'entre elles : l'ajout de nouveaux concepts pour l'expression du parallélisme et la définition d'un langage pivot pour l'algorithmique de bas niveau.

9.2.1 Ajout de stratégies parallèles

Lors de la présentation du langage *HPCML$_n$* dans le chapitre 5, nous avons abordé trois façons d'exprimer du parallélisme au sein d'un modèle : l'exécution de fonctions en concurrence grâce aux concepts `HPCFork` et `HPCJoin`, les boucles parallèles (`HPCParallelFor`) et finalement la décomposition de domaine (`HPCSetEnumerator`). Ces constructions ont été sélectionnées car nous les avons identifiées comme récurrentes au sein des applications développées au *CEA/DAM*.

Cette première sélection de trois constructions est loin d'être un choix définitif. Il semble donc pertinent de s'appuyer sur les travaux concernant les patrons de conception parallèles et les squelettes algorithmiques pour définir de nouveaux concepts parallèles dans le langage *HPCML$_n$*. Une telle diversité permettrait au développeur d'exprimer encore plus de parallélisme dans son application.

Cela étant dit, une trop grande diversité nuirait aussi à l'approche pour plusieurs raisons. Premièrement, un trop grand nombre de concepts nuirait à l'interaction avec les développeurs. On peut citer le cas où deux composants parallèles pourraient être appliqués dans une situation identique : la question de savoir lequel de ces deux composants il faudrait choisir pour avoir les meilleures performances se poserait ? Deuxièmement, il ne faut pas oublier que les concepts parallèles embarquent des stratégies de parallélisme qui doivent être traduites pour chaque architecture cible. Cette traduction est réalisée par des transformations de modèles responsables d'une grande partie de la complexité des générateurs. Une profusion de concepts parallèles aurait un impact important sur le cout de développement d'un générateur pour une architecture donnée. En prenant en compte ces considérations, les futures recherches devront donc déterminer le nombre optimal de concepts parallèles que doit fournir le langage *HPCML$_n$*.

9.2.2 Algorithmique de bas niveau

Nous avons discuté dans la section 6.4 des différentes possibilités pour la gestion de l'algorithmique de bas niveau. Or, la définition d'un langage de type *embedded DSL*, qui serait intégré avec les modèles via le mode de gestion « synchronisation », apparait comme une piste incontournable pour faciliter la définition de transformations sur le contenu des `HPCAlgorithmicContent` et l'exploration des possibilités offertes par la multi-génération.

Ce futur langage devra s'inspirer du langage *Fortran* pour deux raisons principales. Premièrement, le langage *Fortran* est toujours très présent dans la communauté des numériciens. Ils sont donc familiers avec sa syntaxe et la réutilisation d'une partie de cette dernière permettrait de faciliter la transition. Deuxièmement, le langage *Fortran* a été créé pour le développement d'applications de simulation numérique, il contient, par conséquent, de nombreuses constructions adaptées à cette tâche.

Un autre aspect important du langage à prendre en compte concerne l'expression du parallélisme. On viserait ici du parallélisme à grains fins qui exploiterait, par exemple, les capacités vectorielles des processeurs (section 2.2.2). Il faudrait identifier si des constructions génériques ne seraient pas envisageables en addition des constructions permettant de

faire du calcul matriciel.

La définition de la syntaxe concrète d'un tel langage ouvre de nombreuses possibilités de recherche. On pourrait, par , s'inspirer du langage *Fortress* [11] qui propose, entre autres, de se rapprocher du formalisme mathématique en se basant sur deux syntaxes concrètes textuelles : une pour l'édition et une pour la visualisation (voir l'exemple présenté dans la figure 9.1). On peut faire le parallèle avec le langage LaTeX et son mode mathématique. D'un côté, on écrit des équations avec une syntaxe qui peut être facilement tapée au clavier et parsée par le compilateur. De l'autre, on récupère un document où les équations sont affichées avec la notation du formalisme mathématique.

Syntaxe concrète d'édition : `f(x) = x^2 + sin x - cos 2 x`

Syntaxe concrète de visualisation : $f(x) = x^2 + sinx - cos2x$

FIGURE 9.1 – Illustration des deux syntaxes concrètes du langage *Fortress*

9.3 Perspectives pour le langage HPCMLe

Le langage $HPCML_e$ supporte le processus de transformations prenant en entrée les modèles abstraits conformes au langage $HPCML_n$ et générant l'implémentation pour une plateforme donnée. Nous avons vu que le générateur utilisé pour valider le langage $HPCML_n$ possède une structure que l'on a caractérisée de monolithique. Il n'est donc pas possible de paramétrer finement ce processus de transformations à l'aide de modèles. Cette section s'attache à décrire les caractéristiques et les possibilités d'un processus de transformation qui serait fortement paramétrable.

9.3.1 Injection d'optimisations

Les ingénieurs logiciel déterminent au travers de transformations comment les concepts abstraits sont traduits dans une plateforme cible. Nous pensons qu'il serait intéressant que cette traduction soit paramétrable afin de permettre l'application d'optimisations qui ne sont pas déductibles ou valables pour le cas général. L'idée serait que, pour une architecture cible, l'ingénieur logiciel puisse annoter le modèle abstrait (conforme à $HPCML_n$) avec des informations guidant le processus d'optimisations pour cette plateforme. Si l'on prend l'exemple de deux `HPCParallelFor`, le développeur peut vouloir générer un déroulage de boucle pour le premier et pas pour le second.

L'efficacité de l'activité d'optimisation d'un code de simulation, lorsqu'elle est réalisée à l'aveugle, suit généralement une distribution de Pareto [13], c'est-à-dire qu'une petite partie du temps passé à optimiser produit la majorité de l'amélioration des performances. C'est pour cette raison que les développeurs de logiciels de calcul scientifique utilisent des outils de profilage qui leur donnent des indications sur les parties du code les plus gourmandes en ressources afin de concentrer leurs efforts sur ces parties. Malgré sa connaissance de l'architecture cible, l'ingénieur logiciel doit être guidé afin d'améliorer son efficacité. Or, dans notre cas, il annote le modèle pour optimiser le processus de génération, il ne possède donc pas de programme sur lequel il peut utiliser les outils de profilage. Cependant, des solutions sont envisageables pour outrepasser cette limitation. On pourrait, par exemple, générer une version séquentielle et sans optimisation de l'application pour lui appliquer les outils de profilage. Une autre solution serait d'analyser directement les opérations conte-

nues dans le modèle (`HPCAlgorithmicContents` inclus) et d'en déduire une estimation des besoins en ressources de calcul de chaque élément du modèle.

Il est parfois difficile de prévoir l'impact d'une optimisation sur les performances. Par conséquent, il est peut être pertinent, dans certains cas, de vouloir générer plusieurs versions de l'application qui seront basées sur des optimisations différentes pour les évaluer via un banc de test automatique. Cette approche possède toutefois deux limitations qui sont liées. Tout d'abord, la combinatoire des possibilités à explorer peut rapidement devenir exponentielle. Ensuite, la définition d'un micro banc de test, avec des temps d'exécution faibles et représentatif des problèmes traités, est une tâche non triviale.

9.3.2 Vers des codes multi-versionnés

Si on considère que les `HPCAlgorithmicContent` sont définis à l'aide d'un langage synchronisé avec les modèles qui peut être facilement parsé, s'ouvre la possibilité de générer plusieurs versions d'une application de simulation pour une même architecture. Nous abordons dans cette section deux cas pertinents qui méritent d'être investigués.

Version dédiée à une configuration de lancement

En fonction de la taille du problème simulé, un code de simulation peut être exécuté dans différentes configurations de lancement sur une machine. Prenons l'exemple d'un supercalculateur composé d'une interconnexion de plusieurs nœuds à mémoire partagée. Lors de petits calculs, la puissance d'un seul nœud est souvent suffisante, le code va donc s'exécuter sur une architecture à mémoire partagée. Par contre, lors de gros calculs, on va définir une configuration de lancement qui va utiliser plusieurs nœuds. Par conséquent, le code va s'exécuter sur une architecture mixte mélangeant mémoire partagée et mémoire distribuée.

Traditionnellement, on développe des codes adaptables pour qu'ils puissent s'exécuter dans des configurations de lancement diverses. Pourquoi ne pas utiliser la génération multiple pour produire des versions du code de simulation adaptées à une configuration précise de lancement ?

Gestion avancée du parallélisme pour les études paramétriques

Il y a un détail que nous n'avons pas encore clarifié concernant le langage *HPCML*$_n$. Ce dernier fournit des concepts permettant de spécifier des études paramétriques. Cependant, on l'a identifié comme étant dédié au métier de numéricien pour la spécification du modèle mathématique de résolution alors que les études paramétriques sont traditionnellement définies et conduites par l'utilisateur du code de simulation (physicien ou technicien).

Nous avons choisi de définir la paquetage *parametric* dans le langage *HPCML*$_n$ car nous pensons qu'il est pertinent de générer des versions de l'application adaptées à une classe particulière d'études paramétriques afin d'augmenter les performances d'exécution. Actuellement, la principale approche pour conduire une étude paramétrique consiste à lancer plusieurs instances de l'application avec des paramètres différents. Cette approche est donc capable de profiter du parallélisme à gros grains fourni par la machine. Cependant, nous pensons que l'approche *MDE4HPC* peut améliorer l'approche actuelle sur deux points :

- *mutualisation*. Bien que lors de ces études paramétriques, le code de simulation soit lancé avec certains paramètres différents, une part plus ou moins importante des calculs est identique. On pourrait générer des versions qui mutualiserait ces phases de calcul entre les différents scénarios de l'étude paramétrique.

– *vectorisation*. On a vu que l'approche actuelle exploite le parallélisme à gros grains. Nous pensons qu'il serait intéressant de générer des versions tirant partie du parallélisme de type vectoriel. Pour ce faire, il faudrait générer des versions de l'application qui effectueraient leur calcul sur une intervalle de l'étude paramétrique en utilisant des vecteurs plutôt que sur une valeur unique comme pour le code de base. On aurait un mécanisme équivalent au déroulage de boucle, qu'on pourrait appeler déroulage d'études paramétriques, où des vecteurs seraient utilisés à la place des valeurs avec un type simple pour les opérations mathématiques et où les autres types d'instructions dans le code source seraient recopiés autant de fois qu'il y a de valeurs dans l'intervalle spécifié.

9.3.3 Prise en charge du débogage

L'approche *MDE4HPC* permet de réduire le nombre de bogues dans le code généré par le biais de différent mécanismes : contraintes définies par la syntaxe abstraite, typage fort, conception par contrat, génération de l'implémentation des stratégies de parallélisme, etc. Néanmoins, les modèles peuvent toujours contenir des erreurs. Selon l'approche *MDE4HPC*, les numériciens n'ont plus besoin de maitriser les solutions logicielles des plateformes cibles. Dans ces conditions, on ne peut pas leur demander de déboguer du code écrit dans le formalisme de ces solutions logicielles. Des réflexions sur la prise en charge du débogage doivent donc être menées. Est-ce qu'un débogage au niveau du modèle est envisageable ? Sous quelle forme ? Faut-il générer des instructions spécifiquement pour le traitement du débogage ?

9.3.4 Models@run.time

La thématique du *Models@run.time*, c'est-à-dire l'exécution directe des modèles par interprétation plutôt qu'en passant par une étape de génération de code et de compilation, est un sujet recherche en plein essor au sein de la communauté *IDM* [30, 31]. Il ne nous parait pas pertinent, vis-à-vis de l'avancée actuelle des recherches sur le sujet, d'envisager l'exécution directe du modèle de la simulation numérique en phase de production tellement les exigences en termes de performance sont extrêmes dans notre situation.

Cependant, il nous semble intéressant d'explorer l'approche *Models@run.time* comme solutions à deux pistes de recherche que nous venons de présenter dans les sections précédentes. Premièrement, pour guider le processus d'optimisation en fournissant un moyen de raisonner sur les modèles. Deuxièmement, pour permettre la mise en place d'une débogage au niveau du modèle plutôt qu'au niveau du code.

9.4 Discussion

Ce chapitre recense une partie des nombreuses pistes de recherche qu'il reste à explorer dans le cadre de la définition complète de l'approche *MDE4HPC*. Alors que certaines de ces pistes de recherche nécessitent l'application ou l'adaptation de techniques existantes de l'*IDM*, d'autres, au contraire, nécessitent de définir de nouvelles techniques. D'un côté, ces travaux bénéficieraient des techniques de l'*IDM* et, de l'autre, ils y contribueraient.

L'expression populaire *Wife Acceptance Factor* (*WAF*) désigne le niveau moyen de compatibilité d'un objet avec une personne du sexe féminin. À l'image de cette expression, nous proposons la notion de *Developer Acceptance Factor* (*DAF*) qui désigne le niveau moyen de compatibilité entre une « approche logicielle » au sens large et un développeur de la communauté pour laquelle elle a été créée. Nous pensons que la prise en compte de cet

aspect est souvent négligé alors qu'il est un élément clé de l'adoption d'une technologie aussi puissante et efficace soit-elle.

Par exemple, lors de l'introduction du langage *Fortran*, les utilisateurs n'ont pas saisi le sérieux et la puissance de l'approche dès le début et ont continué à développer en langage assembleur [167]. Au bout d'un certain moment, les développeurs ont tout de même constaté que le code généré par le compilateur *Fortran* était aussi bon, voire meilleur que le code écrit à la main. C'est à partir de là que l'adoption du langage fut massive. Cette adoption aurait pu être plus précoce si les promoteurs du langage avaient mieux communiqué ou plus écouté les attentes de la communauté afin d'augmenter le *DAF* du langage *Fortran*.

Il n'a y pas de raisons pour que les développeurs n'aient pas la même méfiance vis-à-vis d'une approche basée sur les modèles comme *MDE4HPC*. Dans ces conditions, et afin de faciliter sa diffusion, il sera donc important, à l'avenir, de réaliser plusieurs études empiriques afin d'adapter l'approche *MDE4HPC* aux attentes de la communauté pour augmenter son *DAF*.

Dans le cadre de ces études, il sera aussi intéressant d'évaluer la capacité du langage à modéliser une grande diversité de domaines physiques. Bien que le langage ait été conçu pour les besoins du *CEA*, rien ne s'oppose en effet, à son utilisation dans un autre cadre.

Chapitre

10

Conclusion

Estimer correctement son degré d'ignorance
est une étape saine et nécessaire.

Hubert Reeves.

L'accélération et la complexification des changements technologiques au niveau des architectures matérielles des supercalculateurs posent de nombreux problèmes aux méthodes de développement actuelles. Le scénario critique étant lorsque le temps nécessaire au portage d'une application pour tirer parti d'une nouvelle architecture matérielle est supérieur à la durée de vie du calculateur.

Dans cette thèse, nous avons proposé une approche basée sur les modèles, nommée *MDE4HPC* (chapitre 4), afin de faciliter le développement et la maintenance d'applications de simulation numérique haute-performance. Cette approche se base sur la définition de modèles abstraits de l'application de simulation qui sont traduits vers les différentes architectures cibles via des transformations de modèles. Au cœur de cette approche se trouve le langage de modélisation *HPCML* qui se décompose en trois couches : $HPCML_p$ (où p signifie *physics*) pour la spécification du modèle physique, $HPCML_n$ (où n signifie *numerical*) pour la spécification du modèle mathématique de résolution et, finalement, $HPCML_e$ (où e signifie *execution*) pour la transformation des modèles abstraits en implémentation. Nous avons défini la syntaxe abstraite et la syntaxe concrète du langage $HPCML_n$ au sein du chapitre 5.

L'approche *MDE4HPC* que nous avons proposée, répond aux trois limitations identifiées dans notre problématique (section 2.1.3) de la manière suivante :

Problème N° 1 **Dépendance vis-à-vis de l'architecture**. C'est grâce à une montée en abstraction que l'approche *MDE4HPC* s'attaque à ce problème. Les applications de simulation numérique sont spécifiées à l'aide de modèles abstraits basés principalement sur le langage $HPCML_n$. Ces modèles sont ensuite traduits par transformations vers différentes plateformes cibles en fonction de l'évolution des ressources matérielles et logicielles de calcul.

Problème N° 2 **Mélange des préoccupations et manque de formalisme**. L'approche *MDE4HPC* propose de séparer les préoccupations au travers des différentes couches du langage *HPCML*. Les aspects (section 5.1.4) ou encore les filtres d'affichages (section 5.2.2) sont, par exemple, un moyen permettant de mettre en œuvre

la séparation des préoccupations métiers. Pour ce qui est du manque de formalisme, le langage *HPCML* répond au problème en unifiant la façon dont sont définis les codes de simulation.

Problème N° 3 **Complexité de programmation**. C'est aussi la montée abstraction réalisée dans le langage *HPCML* qui permet de réduire la complexité de programmation. Un exemple représentatif est l'expression du parallélisme qui repose sur l'utilisation de concepts de haut niveau représentant des stratégies de parallélisme. Le développeur indique comment son code peut être parallélisé mais il n'a pas à sa charge l'implémentation de cette description.

Afin de valider l'approche *MDE4HPC*, nous avons développé un outil nommé *ArchiMDE* (chapitre 6). Ce dernier a été utilisé lors d'un premier développement qui a montré que l'on pouvait générer du code viable en termes de performance et surtout qu'il est possible d'obtenir une réduction des couts de maintenance [168]. Nous avons ensuite procédé à un second développement dans un autre domaine de la physique, l'électromagnétisme, pour estimer la capacité du langage à modéliser des domaines d'application divers (chapitre 7). Finalement l'approche *MDE4HPC* a fait l'objet d'une évaluation se basant sur des principes validés dans la littérature (chapitre 8).

Le chemin vers une approche basée sur les modèles, utilisable dans un contexte industriel, pour le développement d'applications de simulation numérique haute-performance est encore long. Toutefois, l'ensemble des contributions présenté dans cette thèse définit les fondations d'une telle approche et indique des voies possibles pour l'atteinte d'une approche globale.

Annexe

A

Méthodologie d'évaluation des solutions logicielles dédiées à la simulation numérique

Cette annexe présente les cinq propriétés que nous avons définies afin d'évaluer le comportement des solutions logicielles présentées dans l'état de l'art vis-à-vis de notre problématique. Pour chaque propriété, nous avons défini trois niveaux déterminant le degré de complétude de la propriété. Ils possèdent la sémantique suivante :

$$degre\ nul < niveau1 < niveau2 < niveau3 < complet$$

Une précision importante doit être faite : l'évaluation des différentes propriétés n'a pas vocation à apprécier une solution logicielle hors du contexte de cet état de l'art. L'objectif est de pouvoir déterminer une note relative entre plusieurs solutions logicielles pour déterminer lesquelles répondent de manière plus satisfaisantes à notre problématique.

Chamerlain [52] définit un certain nombre de métriques permettant d'évaluer un langage parallèle. Il définit la performance comme métrique première. Ce choix est pertinent étant donné que la performance est une préoccupation majeure de la communauté du calcul haute-performance. Cependant, nous avons délibérément choisi de ne pas évaluer ce critère car il ne nous parait pas concevable de comparer la performance de solutions logicielles aussi diverses. En effet, une telle évaluation supposerait de répondre à de nombreuses questions : Sur quelles machines effectuer les comparaisons ? Avec quel type de développeurs ? Pour quel type d'applications ? De plus, peut-on considérer la performance comme une propriété durable ? Nous partons donc du postulat que l'ensemble des solutions logicielles que nous comparons permettent d'obtenir des performances acceptables.

Les cinq propriétés choisies et les critères permettant de déterminer leur niveau d'atteinte pour une solution logicielle donnée sont les suivants :

- Portabilité : cette propriété définit la capacité d'une solution logicielle à pouvoir utiliser plusieurs plateformes d'exécution.
  1. Solution fortement couplée avec une classe d'architecture donnée, par exemple les architectures de types GPU ;
  2. Solution permettant l'utilisation de plusieurs type d'architectures mais qui possède néanmoins des affinités avec certaines classes d'architectures ;
  3. Solution pouvant être utilisée sur un grand nombre d'architecture et dont la conception permet l'adaptation à d'autres architectures.
- Abstraction/accessibilité : cette propriété définit la facilité de compréhension des concepts d'une solution, notion qui est souvent dépendante du niveau d'abstraction. Cette propriété se rapproche des notions de vue globale et vue locale définies par Chamberlain dans [52]. Dans le cas des solutions logicielles à vue globale, le développeur se concentre sur le comportement de son algorithme métier et délaisse au

compilateur le soin de gérer la distribution des données et les communications. Dans le cas d'une vue locale, les développeurs doivent décrire le comportement qu'aura chaque processus, c'est-à-dire que la gestion et les échanges de données doivent être décrits de façon explicite.

1. Solution fournissant des concepts de très bas niveaux : chargement de données en mémoire, gestion des échanges de données. La solution permet de décrire le « comment » plutôt que le « quoi » de ce que l'on souhaite réaliser. Ce type de solution demande un coût de formation initial élevé et la courbe d'apprentissage est le plus souvent lente ;
2. Solution proposant certains types d'abstraction, notamment pour faciliter l'échange de données ;
3. Solution intégrant des concepts métiers de haut niveaux. Par exemple, ce type de solutions doit fournir directement des primitives pour créer et manipuler des maillages plutôt que d'obliger l'utilisateur à utiliser des tableaux et à projeter l'abstraction dans son esprit. Ce type de solutions permet aux experts métiers d'être opérationnel rapidement.

– Séparation des préoccupations : cette propriété définit le degré d'indépendance des différents aspects d'un code de simulation numérique : physiques / mathématiques / parallélisme / entrées / sorties.

1. Dans ce type de solution non seulement tous les aspects sont mélangés mais aussi fortement corrélés ;
2. Les préoccupations sont mélangés dans ces solutions mais elles n'ont pas d'influence les unes sur les autres ;
3. Ces solutions permettent de séparer les préoccupations en proposant des abstractions à plusieurs niveaux.

– Validation : cette propriété évalue la quantité et la qualité des mécanismes de validation fournis par les solutions logicielles.

1. La solution ne possède pas de concepts ou bibliothèques relatifs au test ou à la validation ;
2. La solution propose un formalisme pour la définition des étapes de tests et validation ;
3. La solution permet d'exprimer des propriétés qui sont vérifiées par de la preuve de programme ou de la génération de tests.

– Communauté/Outils : cette propriété évalue la taille de la communauté d'utilisateur ainsi que la présence d'outils de développement (*IDE*, débogueur, compilateur).

1. Utilisation marginale, ne possède pas la plupart du temps d'outillage spécifique ;
2. Possède plusieurs cas d'utilisation avérés et un outil est disponible ;
3. Utilisation répandue et plusieurs outils similaires sont souvent disponibles.

Table des figures

Acronymes

AGV	Atelier de Génie Logiciel
API	*Application Programming Interface* ou interface de programmation en français
APGAS	*Asynchronous Partitioned Global Address Space* (voir section 3.1.6)
Array-OL	*Array Oriented-Language* (voir section 3.1.13)
BSP	*Bulk Synchronous Parallelism* (voir section 3.1.9)
CAF	*CoArray Fortran* (voir section 3.1.5)
CCA	*Common Component Architecture* (voir section 3.1.8)
ccNUMA	*Cache Coherent NonUniform Memory Access* (voir section 2.2.1)
CDT	*C/*C++ *Development Tooling* http://www.eclipse.org/cdt/
CEA	Commissariat à l'Énergie Atomique et aux Énergies Alternatives
CEA/DAM	Direction des Applications Militaires du *CEA*
Cell BE	*Cell Broadband Engine* (voir section 2.2.2)
CIM	*Computation Independent Model* (voir section 4.1.2)
CPU	*Central Processing Unit* ou unité centrale de traitement en français
CUDA	*Compute Unified Device Architecture* (voir section 3.1.10)
DAF	*Developer Acceptance Factor* (voir section 9.4)
DSL	*Domain-Specific Language* ou langage dédié en français (voir section 3.1.11)
DSML	*Domain-Specific Modeling Language* ou langage de modélisation dédié en français (voir section 4.1.1)
EDP	Équations aux Dérivées Partielles
eDSL	*embedded Domain-Specific Language* ou langage dédié embarqué en français (voir section 3.1.11)
EDF	Électricité de France http://innovation.edf.com/
EFIE	*Electric-Field Integral Equation* ou équation intégrale du champ électrique en français

EMF *Eclipse Modeling Framework* (voir section 6.1)

GMF *Graphical Modeling Framework* (voir section 6.1)

GPU *Graphics Processing Unit* ou processeur graphique en français (voir section 2.2.2)

GWT *Google Web Toolkit*

HLCM *High Level Component Model* (voir section 3.1.8)

HMPP *Hybrid Multicore Parallel Programming* (voir section 3.1.10)

HPCML *High-Performance Computing Modeling Language* (voir chapitre 5)

HPCS *High Productivity Computing Systems* (voir section 3.1.6)

HPF *High Performance Fortran* (voir section 3.1.4)

IDE *Integrated Development Environment* ou environnement de développement intégré en français

IDM Ingénierie Dirigée par les Modèles (voir section 4.1)

LEC Laboratoire Environnement des Codes du CEA/CESTA

MARTE *Modeling and Analysis for Real-Time and Embedded systems*

MDA *Model-Driven Architecture* (voir section 4.1.2)

MDE4HPC *Model-Driven Engineering for High Performance Computing* (voir chapitre 4)

MIMD *Multiple Instruction, Multiple Data* (voir section 2.2.1)

MISD *Multiple Instruction, Single Data* (voir section 2.2.1)

MOF *Meta-Object Facility* (voir section 4.1.2)

MPI *Message Passing Interface* (voir section 3.1.2)

MPP *Massively Parallel Processor* (voir section 2.2.1)

NUMA *NonUniform Memory Access* (voir section 2.2.1)

OCL *Object Constraint Language* (voir section 4.1.2)

OMG *Object Management Group* (voir section 4.1.2)

OSGi *Open Services Gateway initiative* www.osgi.org

OpenMP *Open Multi-Processing* (voir section 3.1.3)

PDM *Platform Description Model* (voir section 4.1.2)

PGAS *Partitioned global address space* (voir section 3.1.5)

PIM *Platform Independent Model* (voir section 4.1.2)

PSM *Platform Specific Model* (voir section 4.1.2)

PTP *Parallel Tools Platform* http://www.eclipse.org/ptp/

PVM *Parallel Virtual Machine* (voir section 3.1.2)

QVT *Query/View/Transformation* (voir section 4.1.2)

SADT *Structured Analysis and Design Technique* [180]

SDK *Software Development Kit* ou kit de développement en français

SIMD *Single Instruction, Multiple Data* (voir section 2.2.1)

SISD *Single Instruction, Single Data* (voir section 2.2.1)

SMP *Symmetric MultiProcessing* (voir section 2.2.1)

SMT *Simultaneous MultiThreading* (voir section 2.2.2)
SPEM *Software Process Engineering Meta-Model* [5]
SPMD *Single Program, Multiple Data* (voir section 3.1.2)
SWT *Standard Widget Toolkit*
SysML *Systems Modeling Language* [114]
TBB *Threading Building Blocks* (voir section 3.1.3)
TCE *Tensor Contraction Engine* (voir section 3.1.11)
UML *Unified Modeling Language* (voir section 4.1.2)
UPC *Unified Parallel C* (voir section 3.1.5)
WAF *Wife Acceptance Factor*
XMI *XML Metadata Interchange* (voir section 4.1.2)
XML *Extensible Markup Language*

Bibliographie personnelle

Conférences

p1 - **Marc Palyart**, David Lugato, Ileana Ober, and Jean-Michel Bruel. 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.

p2 - **Marc Palyart**, David Lugato, Ileana Ober, and Jean-Michel Bruel. MDE4HPC : An Approach for Using Model-Driven Engineering in High-Performance Computing. In Proceedings of the 15th international conference on Integrating System and Software Modeling, SDL'11, pages 245-259. Springer-Verlag, 2011.

p3 - Didier Nassiet, Yohan Livet, **Marc Palyart**, and David Lugato. 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.

Workshops

p4 - David Lugato, **Marc Palyart**, and Christophe Engelvin. Domain-Specific Modeling for Operations Research Simulation in a Large Industrial Context. In The 12th Workshop on Domain-Specific Modeling. ACM, October 2012.

p5 - **Marc Palyart**, David Lugato, Ileana Ober, and Jean-Michel Bruel. HPCML : A Modeling Language Dedicated to High-Performance Scientific Computing. In 1st International Workshop on Model-Driven Engineering for High-Performance and CLoud computing (MDHPCL). ACM, October 2012.

p6 - **Marc Palyart**. HPCML - un langage dédié au calcul scientifique. In Journées sur l'Ingénierie Dirigée par les Modèles (IDM), Lille, pages 31-36, juin 2011.

Revues

p7 - **Marc Palyart**, David Lugato, Ileana Ober, and Jean-Michel Bruel. Le calcul hautes-performances : un nouveau champ d'application pour l'ingénierie de modèles. Génie Logiciel, 97 :41-46, juin 2011.

Poster

Bibliographie

[1] Cea - The Simulation Program. http://www.cea.fr/english_portal/defense/the_simulation_program_lmj_tera_airix.

[2] Gpu applications. http://www.nvidia.com/object/gpu-applications.html.

[3] Intel math kernel library (intel mkl) v11.0. http://software.intel.com/en-us/intel-mkl/.

[4] Matlab. http://www.mathworks.fr/products/matlab/.

[5] Software Process Engineering Meta-Model, version 2.0. Technical report, Object Management Group, 2008. http://www.omg.org/cgi-bin/doc?formal/2008-04-01.

[6] Designing a Digital Future : Federally Funded Research and Development in Networking and Information Technology, 2010. http://www.nitrd.gov/pcast-2010/report/nitrd-program/pcast-nitrd-report-2010.pdf.

[7] The open source developer report 2011 eclipse community survey. Technical report, Eclipse Foundation, 2011. http://www.eclipse.org/org/community_survey/Eclipse_Survey_2011_Report.pdf.

[8] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language : towns, buildings, construction*, volume 2. Oxford University Press, USA, 1977.

[9] Benjamin A. Allan and Rob Armstrong. Ccaffeine Framework : Composing and Debugging Applications Interactively and Running Them Statically. 2005.

[10] Benjamin A. Allan, Robert Armstrong, David E. Bernholdt, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. Mclnnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl.*, 20(2) :163–202, May 2006.

[11] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007. http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.

[12] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack : a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11. IEEE Computer Society Press, 1990.

[13] B.C. Arnold. *Pareto distributions*. Statistical distributions in scientific work series. International Co-operative Publishing House, 1983.

[14] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[15] Matthew H. Austern. *Generic programming and the STL : using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[16] J. Backus. The history of Fortran I, II, and III. *Annals of the History of Computing, IEEE*, 20(4) :68 –78, oct-dec 1998.

[17] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Modern software tools for scientific computing. chapter Efficient management of parallelism in object-oriented numerical software libraries, pages 163–202. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[18] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete-Event System Simulation*. Prentice Hall, 4 edition, December 2004.

[19] Moshe Bar and Maital Neta. Humans prefer curved visual objects. *Psychological Science*, 17(8) :645–648, 2006.

[20] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. Entering the petaflop era : The architecture and performance of roadrunner. In *High Performance Computing, Networking, Storage and Analysis, SC 2008. International Conference for*, 2008.

[21] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin, and J.C. Sancho. A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4), 2008.

[22] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. Understanding the high-performance-computing community : A software engineer's perspective. *IEEE Softw.*, 25(4) :29–36, July 2008.

[23] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klofkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I : Abstract Framework. *Computing*, 82(2–3) :103–119, 2008.

[24] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM : A Grid Extension to Fractal for Autonomous Distributed Components. *Annales des Télécommunications - Annals of Telecommunications*, 2008.

[25] Benoit Baudry. *Assemblage testable et validation de composants*. PhD thesis, Université de Rennes 1, 2003.

[26] G. Baumgartner, A. Auer, D.E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X Gao, R.J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q Lu,

M. Nooijen, R.M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2) :276 –292, feb. 2005.

[27] Kent Beck and Ward Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical Report CR-87-43, Apple Computer, Inc. and Tektronix, Inc., 1987.

[28] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6) :88–95, June 1993.

[29] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Eric Piel, Julien Taillard, and Huafeng Yu. Gaspard2 UML profile documentation. Technical Report RT-0342, INRIA, 2007.

[30] Nelly Bencomo, Gordon Blair, and Robert France. Summary of the workshop models@run.time at models 2006. In *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 227–231. Springer Berlin / Heidelberg, 2007.

[31] Nelly Bencomo, Betty Cheng, Gordon Blair, Robert France, and Cedric Jeanneret. Proceedings of the 6th international workshop on models@run.time at models 2011. volume 794. CEUR-WS, 2011. http://ceur-ws.org/Vol-794/.

[32] Martin Beran. Decomposable bulk synchronous parallel computers. In Jan Pavelka, Gerard Tel, and Miroslav Bartoek, editors, *SOFSEM 99 : Theory and Practice of Informatics*, volume 1725 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1999.

[33] Guntram Berti. *Generic software components for Scientific Computing*. PhD thesis, Technical University of Cottbus, 2000.

[34] Guntram Berti. GrAL – the Grid Algorithms Library. *Future Generation Computer Systems*, 22 :110–122, 2006.

[35] Jean Bézivin. In search of a Basic Principle for Model-Driven Engineering. *Novatica – Special Issue on UML (Unified Modeling Language)*, 5(2), 2004.

[36] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, 2001.

[37] Julien Bigot and Christian Pérez. High Performance Composition Operators in Component Models. In Lucio Grandinetti Gerhard R. Joubert Ian Foster, Wolfgang Gentzsch, editor, *High Performance Computing : From Grids and Clouds to Exascale*, volume 20 of *Advances in Parallel Computing*, pages 182 – 201. IOS Press, 2011.

[38] Rob H. Bisseling. *Parallel Scientific Computation : A Structured Approach Using BSP and MPI*. Oxford University Press, 2004.

[39] X. Blanc. *MDA en action : Ingénierie logicielle guidée par les modèles*. Eyrolles, 2005.

[40] Francois Bodin and Stephane Bihan. Heterogeneous multicore parallel programming for graphics processing units. *Sci. Program.*, 17(4) :325–336, December 2009.

[41] Markus Bornemann, Rob V. van Nieuwpoort, and Thilo Kielmann. MPJ/Ibis : a flexible and efficient message passing platform for Java. In *Proceedings of the 12th*

European PVM/MPI users' group conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, PVM/MPI'05, pages 217–224. Springer-Verlag, 2005.

[42] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Journal*, 2(4) :27–66, 1997.

[43] Walter Brenner, H. Wittig, and Rudiger Zarnekow. *Intelligent Software Agents : Foundations and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[44] N. Brookwood. AMD fusion family of APUs : Enabling a superior immersive pc experience. Technical report, Insight 64, 2010.

[45] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[46] François Cantonnet, Yiyi Yao, Mohamed M. Zahran, and Tarek A. El-Ghazawi. Productivity analysis of the UPC language. In *IPDPS*, 2004.

[47] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00. IEEE Computer Society, 2000.

[48] Jeffrey C. Carver. Report from the second international workshop on software engineering for computational science and engineering (SE-CSE 09). *SIGSOFT Softw. Eng. Notes*, 34(5) :48–51, October 2009.

[49] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3) :291–312, 2007.

[50] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[51] Bradford L. Chamberlain, Sung eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL's WYSIWYG performance model. In *In Third International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 50–61. IEEE Computer Society Press, 1998.

[52] Bradford Lee Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD thesis, University of Washington, 2001.

[53] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan and Kaufmann, San Diego, CA, 2001.

[54] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 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, pages 519–538. ACM, 2005.

[55] Eric Clayberg and Dan Rubel. *Eclipse : Building Commercial-Quality Plug-ins (2nd Edition) (Eclipse)*. Addison-Wesley Professional, 2006.

[56] Norman S. Clerman and Walter Spector. *Modern Fortran : Style and Usage*. Cambridge University Press, 2011.

[57] Murray Cole. *Algorithmic skeletons : structured management of parallel computation*. Research monographs in parallel and distributed computing. Pitman, London, 1989.

[58] Murray Cole. Bringing skeletons out of the closet : a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3) :389–406, March 2004.

[59] David L. Colton and Rainer Kress. *Integral equation methods in scattering theory.* Wiley, 1983.

[60] Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific development with visual studio DSL tools.* Addison-Wesley Professional, first edition, 2007.

[61] Leonardo Dagum and Ramesh Menon. OpenMP : An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5 :46–55, 1998.

[62] Vincent Danjean and Raymond Namyst. Controling Kernel Scheduling from User Space : an Approach to Enhancing Applications' Reactivity to I/O Events. In *Proceedings of the 2003 International Conference on High Performance Computing (HiPC 03)*, 2003.

[63] Antonio Wendell De Oliveira Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Technical Report RR-7525, INRIA, February 2011.

[64] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J.C. Dufourd, and J.L. Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multidimensionnel. In *15 Colloque sur le traitement du signal et des images, FRA, 1995.* GRETSI, Groupe d'Etudes du Traitement du Signal et des Images, 1995.

[65] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 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, pages 9 :1–9 :12. ACM, 2011.

[66] Daniele Antonio Di Pietro, Jean-Marc Gratien, and Christophe PrudHomme. A domain-specific embedded language in C++ for lowest-order discretizations of diffusive problems on general meshes. December 2011.

[67] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical report, 2009.

[68] Isabel Dietrich, Falko Dressler, Volker Schmitt, and Reinhard German. SYNTONY : network protocol simulation based on standard-conform UML 2 models. In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ValueTools '07. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[69] Isabel Dietrich, Reinhard German, Harald Koestler, and Ulrich Ruede. Modeling Multigrid Algorithms for Variational Imaging. In *21st Australian Software Engineering Conference (ASWEC 2010)*, pages 224–234, Auckland, New Zealand, April 2010. IEEE.

[70] Edsger W. Dijkstra. Cooperating sequential processes. 1968.

[71] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[72] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, J.C. Andre, David Barkai, J.Y. Berthou, Taisuke Boku, Bertrand Braunschweig, and

Others. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1) :3, 2011.

[73] Jack Dongarra, Robert Graybill, William Harrod, Robert F. Lucas, Ewing L. Lusk, Piotr Luszczek, Janice McMahon, Allan Snavely, Jeffrey S. Vetter, Katherine A. Yelick, Sadaf R. Alam, Roy L. Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy S. Meredith, and Mustafa M. Tikir. DARPA's HPCS Program-History, Models, Tools, Languages. *Advances in Computers*, 72 :1–100, 2008.

[74] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. 2003.

[75] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 15. IEEE, 2004.

[76] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL : Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 2011.

[77] Gábor Dózsa, Tibor Fadgyas, and Péter Kacsuk. GRAPNEL : A Graphical Programming Language for Parallel Programs. In *In proc. uP'94 : The Eighth Symposium on Microcomputer and Microprocessor Applications*, pages 304–314. IEEE Press, 1994.

[78] N. Elliot, S. Kohn, and B. Smolinski. Language interoperability for high-performance parallel scientific components. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 1999), San Francisco, CA*, 1999.

[79] Ralf S. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. In *In Proc. USENIX Tech. Conf*, pages 239–250, 2000.

[80] Tom Epperly, Scott R. Kohn, and Gary Kumfert. Component technology for high-performance scientific simulation software. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 69–86. Kluwer, B.V., 2001.

[81] Moritz Eysholdt, Sören Frey, and Wilhelm Hasselbring. EMF Ecore Based Meta Model Evolution and Model Co-Evolution. 29(2) :20–21, 2009. Proceedings of the 11th Workshop Software-Reengineering (WSR 2009).

[82] Robert Falgout and Ulrike Yang. HYPRE : A Library of High Performance Preconditioners. In Peter Sloot, Alfons Hoekstra, C. Tan, and Jack Dongarra, editors, *Computational Science ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer Berlin / Heidelberg, 2002.

[83] Jean-Marie Favre. Foundations of Meta-Pyramids : Languages vs. Metamodels - Episode II : Story of Thotus the Baboon1. In *Language Engineering for Model-Driven Software Development*, 2004.

[84] Jean-Marie Favre. Languages evolve too ! changing the software time scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 33–44. IEEE Computer Society, 2005.

[85] Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, 2006.

[86] Wu-Chun Feng and Tom Scogland. The Green500 List : Year One. In *5th IEEE Workshop on High-Performance, Power-Aware Computing (in conjunction with the 23rd International Parallel & Distributed Processing Symposium)*, Rome, Italy, May 2009.

[87] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In *Model Driven Engineering Languages and Systems*, 2007.

[88] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948 –960, sept. 1972.

[89] MPI Forum. MPI : A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[90] Robert France and Bernhard Rumpe. Model-driven development of complex software : A research roadmap. In *Future of Software Engineering*, FOSE '07, pages 37–54. IEEE Computer Society, 2007.

[91] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML : Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[92] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, 2005.

[93] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI : Goals, concept, and design of a next generation MPI implementation. In *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

[94] David Geer. Industry trends : Chip makers turn to multicore processors. *Computer*, 38, 2005.

[95] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM : Parallel virtual machine : a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994.

[96] S. Gérard and B. Selic. The UML MARTE Standardized Profile. In *Proceedings of the 17th IFAC World Congress*, pages 6–11, 2008.

[97] H.H. Goldstine, J. Von Neumann, H. Mathematician, J. von Neumann, and J. von Neumann. Planning and coding of problems for an electronic computing instrument. 1947.

[98] C. Gómez. *Engineering and Scientific Computing with SciLab*. Birkhauser, 1999.

[99] Jean Gonnord, Pierre Leca, and François Robin. Au delà de 50 mille milliards d'opérations par seconde ! *La Recherche*, N. 393, January 2006.

[100] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks : high-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12) :1135–1160, November 2010.

[101] Jean-Marc Gratien. Implementing a domain-specific embedded language for lowest-order variational methods with boost proto. *C++Now*, May 2012.

[102] Jack Greenfield and Keith Short. Software factories : assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03. ACM, 2003.

[103] Shirley Gregor. The nature of theory in information systems. *Management Information Systems Quarterly*, 30(3) :611–642, September 2006.

[104] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6) :789–828, 1996.

[105] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2nd ed.) : portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999.

[106] Gilles Grospellier and Benoit Lelandais. The Arcane development framework. In *POOSC '09*. ACM, 2009.

[107] Object Management Group. Corba component model 4.0 specification. Technical report, Object Management Group, 2006.

[108] Object Management Group. MOF 2.4 Specification. Technical report, 2011. http://www.omg.org/spec/MOF/2.4/.

[109] Object Management Group. OMG Query/View/Transformation (OMG QVT), V1.1. Technical report, 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

[110] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.4. Technical report, 2011. http://www.omg.org/spec/UML/2.4/Infrastructure/PDF.

[111] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.4. Technical report, 2011. http://www.omg.org/spec/UML/2.4/Superstructure/PDF.

[112] Object Management Group. OMG XML Metadata Interchange (OMG XMI), V2.4.1. Technical report, 2011. http://www.omg.org/spec/XMI/2.4.1/PDF/.

[113] Object Management Group. OMG Object Constraint Language (OMG OCL), V2.3.1. Technical report, 2012. http://www.omg.org/spec/OCL/2.3.1/PDF/.

[114] Object Management Group. OMG Systems Modeling Language (SysML), V1.3. Technical report, 2012. http://www.omg.org/spec/SysML/1.3/PDF.

[115] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. Hybrid bulk synchronous parallelism library for clustered SMP architectures. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 55–62. ACM, 2010.

[116] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. A framework for an automatic hybrid MPI+OpenMP code generation. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, pages 48–55. Society for Computer Simulation International, 2011.

[117] Robert Hanson and Adam Tacy. *GWT in Action : Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.

[118] R.F. Harrington. *Time-Harmonic Electromagnetic Fields*. IEEE Press Series on Electromagnetic Wave Theory. Wiley, 2001.

[119] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00. IEEE Computer Society, 2000.

[120] Markus Herrmannsdoerfer. COPE - A Workbench for the Coupled Evolution of Metamodels and Models. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin / Heidelberg, 2011.

[121] M.D. Hill, N.P. Jouppi, and G. Sohi. *Readings in computer architecture*. Morgan Kaufmann Pub, 2000.

[122] S Hitchman. The details of conceptual modelling notations are important - a comparison of relationship normative language. *Communications of the Association for Information Systems*, 9, 2002.

[123] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.

[124] Jean-Marc Jezequel, Michel Train, and Christine Mingins. *Design Patterns with Contracts*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[125] Shoaib Kamil, Derrick Coetzee, Scott Beamer, Henry Cook, Ekaterina Gonina, Jonathan Harper, Jeffrey Morlan, and Armando Fox. Portable parallel performance from sequential, productive, embedded domain-specific languages. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 303–304. ACM, 2012.

[126] Steven Kelly, Kalle Lyytinen, and Matti Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In Panos Constantopoulos, John Mylopoulos, and Yannis Vassiliou, editors, *Advanced Information Systems Engineering*, volume 1080 of *Lecture Notes in Computer Science*, pages 1–21. Springer Berlin / Heidelberg, 1996.

[127] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance Fortran. *Commun. ACM*, 54(11) :74–82, November 2011.

[128] Jeremy Kepner. HPC Productivity : An Overarching View. *Int. J. High Perform. Comput. Appl.*, 18(4) :393–397, November 2004.

[129] K. Keutzer and T. Mattson. Our Pattern Language (OPL) : A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, 2009.

[130] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. 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.

[131] KhronosGroup. The OpenCL specification v1.2. Technical report, 2011. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[132] David Kirk. NVIDIA CUDA software and GPU parallel computing architecture. In *ISMM*, pages 103–104, 2007.

[133] Anneke G. Kleppe. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering*, Grenoble, France, 2007.

[134] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003.

[135] C.H. Koelbel. *The High Performance Fortran handbook*. The MIT press, 1994.

[136] Harald Köstler. *A multigrid framework for variational approaches in medical image processing and computer vision*. PhD thesis, 2008.

[137] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2), march-april 2003.

[138] Thomas Kühne. What is a model ? In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[139] Leslie Lamport. *LATEX : a document preparation system : user's guide and reference manual*. Addison-Wesley Pub. Co., 1994.

[140] Frédéric Loulergue. Parallel superposition for bulk synchronous parallel ML. In *Proceedings of the 2003 international conference on Computational science : PartIII*, ICCS'03, pages 223–232. Springer-Verlag, 2003.

[141] Jochen Ludewig. Models in software engineering : an introduction. *Software and Systems Modeling*, 2 :5–14, 2003. 10.1007/s10270-003-0020-3.

[142] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU : general purpose computation on graphics hardware. In *Annual Conference on Computer Graphics*, 2004.

[143] David Lugato. Model-driven engineering for high-performance computing applications. In *The 19th IASTED International Conference on Modelling and Simulation*, Quebec City, Quebec, Canada, May 2008.

[144] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55. ACM, 2009.

[145] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pages 23–. IEEE Computer Society, 2002.

[146] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steven Bromling. Generating parallel program frameworks from parallel design patterns. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 95–104. Springer-Verlag, 2000.

[147] B.L. Massingill, T.G. Mattson, B.A. Sanders, et al. A pattern language for parallel application programming. In *Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP 1999)*, 1999.

[148] MATLAB. *version 8 (R2012b)*. The MathWorks Inc., 2012.

[149] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[150] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004.

[151] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations. In *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[152] John Merlin and Anthony Hey. An introduction to high performance fortran. *Sci. Program.*, 4(2) :87–113, April 1995.

[153] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Survey*, 37(4) :316–344, December 2005.

[154] Hans W. Meuer. The TOP500 Project : Looking Back Over 15 Years of Supercomputing Experience. *Informatik-Spektrum*, 31(3) :203–222, June 2008.

[155] Bertrand Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.

[156] George Miller. The magical number seven, plus or minus two : Some limits on our capacity for processing information. *The Psychological Review*, 63 :81–97, 1956.

[157] Daniel L. Moody. The "physics" of notations : Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35 :756–779, 2009.

[158] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8) :114–117, April 1965.

[159] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11 – 13, 1975.

[160] Pierre-Alain Muller, Frédéric Fondement, and Benoit Baudry. Modeling Modeling. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, 2009.

[161] Didier Nassiet, Yohan Livet, Marc Palyart, and David Lugato. 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.

[162] Peter Newton and James C. Browne. The CODE 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 167–177. ACM, 1992.

[163] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays : A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10 :169–189, 1996.

[164] S. Northover and M. Wilson. *SWT : the standard widget toolkit*. The Eclipse series. Addison-Wesley, 2004.

[165] United States. Dept. of Defense. COBOL : Initial specifications for a common business oriented language, 1960.

[166] Hervé Oudin. Méthode des éléments finis. http://cel.archives-ouvertes.fr/cel-00341772/PDF/bouquin.pdf, September 2008.

[167] David Padua. The fortran I compiler. *Computing in Science and Engineering*, 2 :70–75, 2000.

[168] Marc Palyart, David Lugato, Ileana Ober, and Jean-Michel Bruel. 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.

[169] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen mei W. Hwu. FCUDA : Enabling efficient compilation of CUDA kernels onto FPGAs. In *SASP*, pages 35–42, 2009.

[170] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12) :1326–1335, December 1985.

[171] S. V. Pennington and M. Berzins. New NAG library software for first-order partial differential equations. *ACM Trans. Math. Softw.*, 20(1) :63–99, 1994.

[172] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41, 2006.

[173] S. Pllana and T. Fahringer. Performance prophet : A performance modeling and prediction tool for parallel and distributed programs. In *Parallel Processing, 2005. ICPP 2005 Workshops. International Conference Workshops on*, pages 509–516. IEEE, 2005.

[174] Sabri Pllana, Siegfried Benkner, Eduard Mehofer, Lasse Natvig, and Fatos Xhafa. Euro-par 2008 workshops - parallel processing. chapter Towards an Intelligent Environment for Programming Multi-core Computing Systems, pages 141–151. Springer-Verlag, 2009.

[175] Sabri Pllana and Thomas Fahringer. On customizing the UML for modeling performance-oriented applications. In Jean-Marc Jezequel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 83–102. Springer Berlin / Heidelberg, 2002.

[176] John Reid. Coarrays in the next fortran standard. *SIGPLAN Fortran Forum*, 29(2), 2010.

[177] J. Reinders. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[178] W. Rodrigues, F. Guyomarc'h, and J. Dekeyser. An mde approach for automatic code generation from uml/marte to opencl. *Computing in Science Engineering*, (99) :1, 2012.

[179] Tiark Rompf and Martin Odersky. Lightweight modular staging : a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[180] D. T. Ross and K. E. Schoman, Jr. Classics in software engineering. chapter Structured analysis for requirements definition, pages 363–386. Yourdon Press, 1979.

[181] G. Sadaka. FreeFem++, a tool to solve PDEs numerically. *Arxiv preprint arXiv :1205.1293*, 2012.

[182] S. Saini, Haoqiang Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *High Performance Computing (HiPC), 2011 18th International Conference on*, 2011.

[183] Jason Sanders and Edward Kandrot. *CUDA by Example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[184] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. Technical report, Toronto, Canada, 2010.

[185] Max Schulz. The end of the road for silicon ? *Nature*, 399(6738) :729–730, 1999.

[186] Judith Segal. When software engineers met research scientists : A case study. *Empirical Softw. Engg.*, 10(4) :517–536, October 2005.

[187] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5), 2003.

[188] Bran Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19 – 25, 2003.

[189] A. Shet, V. Tipparaju, and R. Harrison. Asynchronous programming in UPC : A case study and potential for improvement. In *First Workshop on asynchrony in the PGAS model*, 2009.

[190] Keng Siau. Information modeling and method engineering : a psychological perspective. *Journal of Database Management*, 10(4) :44–50, December 1999.

[191] David B. Skillicorn, Jonathan M. D. Hill, and W. F. Mccoll. Questions and answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[192] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[193] Richard Soley. Model driven architecture. Technical report, Object Management Group (OMG), 2000.

[194] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[195] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.

[196] J.E. Stone, D. Gohara, and G. Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3) :66, 2010.

[197] V. S. Sunderam. PVM : A Framework for Parallel Distributed Computing. *Concurrency : Practice and Experience*, 2 :315–339, 1990.

[198] Marc Snir Susan L. Graham and Committee on the Future of Supercomputing National Research Council Cynthia A. Patterson, Editors. *Getting Up to Speed : The Future of Supercomputing*. The National Academies Press, 2004.

[199] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4), apr 1997.

[200] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[201] Julien Taillard, Frederic Guyomarc'h, and Jean-Luc Dekeyser. A Graphical Framework for High Performance Computing Using An MDE Approach. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 165–173, Washington, DC, USA, 2008. IEEE Computer Society.

[202] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik, and Steve MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 203–215. ACM, 2003.

[203] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading : Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, 1995.

[204] 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.

[205] Mateo Valero. A european perspective on supercomputing. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 1–1. ACM, 2009.

[206] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[207] R.A. Van de Geijn. *Using PLAPACK–parallel linear algebra package*. The MIT Press, 1997.

[208] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6), June 2000.

[209] Andres Varga. The OMNeT++ Discrete Event Simulation System. *Proceedings of the European Simulation Multiconference (ESM'2001)*, June 2001.

[210] John Von Neumann. First draft of a report on the EDVAC. Technical report, Los Alamos National Laboratory, 1945.

[211] W3C. Mathematical Markup Language (MathML), V3.0 . Technical report, 2010. http://www.w3.org/TR/MathML3/mathml.pdf.

[212] G.R. Watson and N.A. DeBardeleben. Developing scientific applications using Eclipse. *Computing in Science Engineering*, 8(4) :50 –61, july-aug 2006.

[213] Michèle Weiland. 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.

[214] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31 :50–59, 2011.

[215] Keming Zhang, Kostadin Damevski, and Steven G. Parker. SCIRun2 : A CCA framework for high performance computing. In *In Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, pages 72–79. IEEE Press, 2004.

A Model-Based Approach for the Development of High-Performance Scientific Computing Software

Abstract : The development and maintenance of high-performance scientific computing software is a complex task. This complexity results from the fact that software and hardware are tightly coupled. Furthermore current parallel programming approaches lack of accessibility and lead to a mixing of concerns within the source code. In this thesis we define an approach for the development of high-performance scientific computing software which relies on model-driven engineering. In order to reduce both duration and cost of migration phases toward new hardware architectures and also to focus on tasks with higher added value this approach called *MDE4HPC* defines a domain-specific modeling language. This language enables applied mathematicians to describe their numerical 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 with model transformations that can be shared among several software developments. To evaluate the effectiveness of this approach we developed a tool called *ArchiMDE*. Using this tool we developed different numerical simulation software to validate the design choices made regarding the modeling language.

Auteur : Marc PALYART-LAMARCHE

Une approche basée sur les modèles pour le développement d'applications de simulation numérique haute-performance

Directeurs de thèse : Jean-Michel BRUEL, Ileana OBER, David LUGATO

Lieu et date de soutenance : Institut Lasers et Plasmas (ILP) 33114 Le Barp, le 18 décembre 2012

Résumé. Le développement et la maintenance d'applications de simulation numérique haute-performance sont des activités complexes. Cette complexité découle notamment du couplage fort existant entre logiciel et matériel ainsi que du manque d'accessibilité des solutions de programmation actuelles et du mélange des préoccupations qu'elles induisent. Dans cette thèse nous proposons une approche pour le développement d'applications de simulation numérique haute-performance qui repose sur l'ingénierie des modèles. Afin à la fois de réduire les couts et les délais de portage sur de nouvelles architectures matérielles mais également de concentrer les efforts sur des interventions à plus haute valeur ajoutée, cette approche nommée *MDE4HPC*, définit un langage de modélisation dédié. Ce dernier permet aux numériciens de décrire la résolution de leurs modèles théoriques dans un langage qui d'une part leur est familier et d'autre part est indépendant d'une quelconque architecture matérielle. Les différentes préoccupations logicielles sont séparées grâce à l'utilisation de plusieurs modèles et de plusieurs points de vue sur ces modèles. En fonction des plateformes d'exécution disponibles, ces modèles abstraits sont alors traduits en implémentations exécutables grâce à des transformations de modèles mutualisables entre les divers projets de développement. Afin de valider notre approche nous avons développé un prototype nommé *ArchiMDE*. Grâce à cet outil nous avons développé plusieurs applications de simulation numérique pour valider les choix de conception réalisés pour le langage de modélisation.

Mots clés : Ingénierie dirigée par les modèles, calcul haute-performance, simulation numérique, langage de modélisation dédié

Discipline administrative : Informatique

Unité de recherche : Institut de Recherche en Informatique de Toulouse (IRIT, UMR 5505) / Commissariat à l'Énergie Atomique et aux Énergies Alternatives - Centre d'Études Scientifiques et Techniques d'Aquitaine (CEA/CESTA)