Subvention accordée par la Région wallonne
à une unité de recherche universitaire
ou de niveau universitaire

Open Numerical Engineering LABoratory

ONELAB

Rapport technique

N° 1

période du 01/10/2010 au 30/09/2011

Numéros de convention

WIST 3.0 No 1017086

Note : ce rapport technique est un document évolutif. Il sera complété au fur et à mesure de l'avancement du projet.

Introduction

Ce rapport technique reprend les résultats principaux du projet ONELAB, dont l'objectif est de développer une plateforme d'intégration de logiciels libres pour le calcul scientifique en ingénierie.

Motivation générale et contexte économique

Les entreprises en Région Wallonne sont grandes utilisatrices de logiciels de calcul scientifique. Parmi les logiciels les plus utilisés, on peut citer Fluent pour la mécanique des fluides, Abaqus, Ansys ou Samcef pour le calcul de structures, et Flux2D/3D ou Ansoft pour la simulation électromagnétique. Le coût des licences de ces logiciels commerciaux peut s'élever à plusieurs dizaines de milliers d'euros par an. Si cet investissement est justifié et supportable par des entreprises ayant un besoin très régulier de simulations et disposant d'un personnel qualifié les utilisant de manière régulière, il sort très rapidement du budget d'une PME n'ayant besoin de recourir à la simulation que de manière sporadique. Les PMEs peuvent alors se tourner vers des bureaux d'études comme GDTech. Malheureusement, pour des simulations complexes, le coût des licences encouru par le bureau d'étude, répercuté sur la PME, devient également très rapidement prohibitif. Pour fixer les idées, une licence Fluent pour une station de travail bi-processeur est de l'ordre de 30.000 euros/an. Une PME ne peut donc habituellement pas se permettre un calcul de grande taille nécessitant quelques dizaines de processeurs... une situation aberrante en 2010, à l'heure où une simple station de travail possède 8 coeurs de calcul.

Les logiciels open source constituent une solution à ce problème. En effet, la communauté du logiciel libre produit depuis plusieurs années des logiciels de niveau professionnel. Un exemple en bureautique est la suite OpenOffice, de plus en plus répandue. Dans le domaine du calcul scientifique, la plupart des distributions Linux proposent également des logiciels scientifiques de grande qualité. Certains sont des librairies de bas niveau, réservées à un cercle de spécialistes (PETSc, Metis, VTK, MadLib, Open CASCADE, etc.). Mais il existe également des logiciels de haut niveau destinés à l'utilisateur final, par exemple pour des problématiques de segmentation d'images (ITK), de création de maillage éléments finis (Gmsh, Meshlab) ou de visualisation des résultats (Paraview, Gmsh). Dans les domaines de la mécanique des solides, de la mécanique des fluides et de l'électromagnétisme, la qualité de ces logiciels de haut niveau (Code_Aster en mécanique du solide, OpenFOAM en mécanique des fluides, GetDP en électromagnétisme) est telle qu'ils sont compétitifs par rapport aux solutions commerciales classiques, tant au niveau de leurs capacités que de leurs performances, tandis qu'il sont souvent techniquement supérieurs, plus ouverts et flexibles. Cependant, malgré leurs qualités reconnues, ces logiciels restent très peu utilisés par les PMEs wallonnes. Nous pensons que l'absence d'une interface facile d'emploi pour le pré- et le post-traitement est en grande partie responsable de cet état de fait, de même que, pour les logiciels émanant du monde académique, un manque de documentation et d'exemples adaptés. Un frein à l'adoption de solutions open source réside également dans l'amalgame qui perdure dans le monde industriel entre « logiciel open source » et « freeware » (logiciel « gratuit », « limité », « non-professionnel »).

Forts de notre expertise dans le domaine du logiciel libre (Gmsh, Madlib, GetDP), nous

souhaitons dans ce projet développer un outil logiciel qui permette de piloter divers logiciels libres de qualité professionnelle (OpenFOAM, Code_Aster, GetDP, Elmer) via l'unification de la gestion du pré- et du post-traitement. L'interface unique rendra l'accès à une plateforme libre beaucoup plus aisé pour les PME wallonnes, diminuant les coûts liés à l'achat de licences de logiciels propriétaires et augmentant donc leur compétitivité. Cette interface unique permettra également de mettre sur pied une plateforme très utile au niveau pédagogique, tant dans les Universités que dans les écoles supérieures et les centres de formation continuée. La formation à ces nouvelles technologies open source permettra à nouveau de diminuer les coûts associés à l'achat d'onéreuses licences des produits commerciaux mais aussi à préparer les futurs ingénieurs aux outils qu'ils rencontreront au sein des PMEs wallonnes une fois diplômés.

Motivation technique

Une des principales difficultés rencontrées par les ingénieurs non-spécialistes des méthodes de simulation et souhaitant exploiter des logiciels libres est l'**hétérogénéité des outils** d'une chaîne de résolution open source (en contraste avec le haut niveau d'intégration offert par les outils commerciaux équivalents). Une chaîne de résolution complète comprend en général les 5 étapes suivantes:

- Installation des logiciels
- Création ou import de géométrie dans un logiciel de CAO (Gmsh, Ansys, etc.)
- Maillage (Gmsh, Hypermesh, Ansys, etc.)
- Calcul dans l'environnement de simulation (GetDP, OpenFoam, Code_Aster, dg, etc.)
- Post-traitement ("post-processing") des résultats (Gmsh, Paraview, etc.)

Lorsque l'on prend le parti de travailler avec des logiciels libres, on est amené à devoir intégrer dans cette chaîne de résolution plusieurs logiciels indépendants. Cette hétérogénéité des outils logiciels va de pair avec une hétérogénéité des concepts utilisés, des formats de données, etc., ce qui rend la tâche en général difficile.

Une seconde difficulté, tout aussi importante, est l'absence d'une formalisation explicite (autrement que par essai et erreur) des relations existantes entre les paramètres d'entrée d'un modèle (c.-à-d. les informations qui sont à la disposition de l'utilisateur) et les caractéristiques de la chaîne de résolution particulière qui permet d'obtenir une solution satisfaisante. Cette formalisation fait l'objet d'un niveau d'abstraction spécifique, qu'on peut appeler **couche "experte"** (c.-à-d. relevant du domaine de l'expertise) qui représente l'essentiel de la valeur ajoutée des logiciels commerciaux, ainsi que la clé *sine qua non* d'accès pour l'utilisateur non-spécialiste. C'est précisément l'absence de cette couche experte formalisée sitôt que l'on se tourne vers un ensemble d'outils de simulations indépendants plutôt que vers un environnement intégré, tel que ceux offerts par les logiciels de simulation commerciaux, qui représente l'obstacle majeur à la diffusion des logiciels libres dans le domaine de l'ingénierie industrielle. La composante "experte" est généralement fournie comme un produit fini. Il s'agit d'un ensemble de fonctionnalités prédéfinies, matérialisées par un système de menus et permettant de sélectionner des schémas de modélisation préétablis et validés. Selon ce paradigme, la robustesse est "garantie" de façon intrinsèque, et c'est elle précisément qui représente la valeur ajoutée du produit et justifie son prix.

Dans le cas de logiciels libres par contre, sachant que l'on vise à la combinaison de logiciels indépendants, inconnus a priori, il est logiquement impossible d'assurer aucune validation préalable des schémas de résolution, puisque les fonctionnalités disponibles dépendent entièrement du choix des logiciels utilisés. La composante "experte" doit donc nécessairement reposer sur un principe complètement différent, plutôt de nature extrinsèque cette fois-ci, ainsi qu'il sera montré plus loin.

C'est à cette double question (hétérogénéité des outils, absence d'une couche "experte") que le projet ONELAB vise à apporter une solution.

Principes généraux de la mise en oeuvre

Les principes généraux qui sont à la base de l'élaboration de l'interface ONELAB sont au nombre de 5. Ils sont maintenant détaillés.

Interface basée sur une triple abstraction

Les interactions entre les différents maillons d'une chaîne de résolution se font essentiellement par l'échange de fichiers de données et de paramètres. Pour pallier l'hétérogénéité des outils utilisés, il faut donc créer une interface abstraite et persistante capable de gérer ces interactions de façon centralisée et transparente pour l'utilisateur. Les différents maillons de la chaîne de résolution sont en effet amenés à partager un ensemble de paramètres communs, lesquels doivent être introduits aux différents niveaux sous des formats spécifiques.

Le rôle de l'interface abstraite est de centraliser l'ensemble des paramètres du modèle dans une structure de données unique, et de les distribuer ensuite aux différents modules. L'interface abstraite donne ainsi à la chaîne de résolution hétérogène l'aspect extérieur d'un modèle unique. C'est le niveau d'abstraction recherché par l'utilisateur. C'est aussi celui qui permet de contrôler de façon centralisée les interactions entre les différents modules de la chaîne de résolution.

L'interface abstraite ONELAB est implantée dans le logiciel open source Gmsh. Elle est basée sur une triple abstraction :

1. abstraction de l'interface vers les modeleurs géométriques (CAO) et la génération/ simplification de maillages ;
2. abstraction de la définition des propriétés physiques, des contraintes et des paramètres de pilotage des différents logiciels utilisés ;
3. abstraction et consolidation des fonctionnalités de post-traitement.

Approche client-serveur

L'approche choisie pour mettre en oeuvre la fonctionnalité d'intégration de ONELAB est celle d'une relation client-serveur : l'ensemble des paramètres sont stockés sur un serveur vis-à-vis duquel les logiciels de simulation, ainsi que l'utilisateur le cas échéant, jouent le rôle des clients. Une approche de type client-serveur permet de considérer de façon naturelle et transparente les différents cas de figure où les clients tournent soit sur la même machine soit sur des machines différentes.

Implémentation en C++

Pour une portabilité la plus large possible (y compris vers les plateformes de type iPad), la

librairie de gestion client-serveur est écrite en C++ plutôt que dans un langage interprété. Dans un second temps, les fonctionalités C++ seront rendues utilisables au départ d'un environnement de calcul scientifique interprété comme Python ou Matlab, de façon à offrir à l'utilisateur la souplesse d'un langage interprété. Dans le principe, cependant, la version compilée (C++) et la version interprétée (Python, Matlab) de ONELAB sont identiques. Cette approche permet d'éviter le développement d'un parser ONELAB spécifique, puisque dans tous les cas (C++, Python ou Matlab), on pourra se reposer sur un parser existant.

Fonctionnement générique

L'implémentation de la relation client-serveur est conçue pour fonctionner de façon totalement générique, c'est-à-dire qu'elle peut utiliser les logiciels clients tels quels, au moyen de leurs canaux de communications (entrées-sorties) propres. Lorsque le logiciel client n'intègre pas directement l'interface ONELAB (client "encapsulé"), une couche interface est ajoutée pour permettre la communication du client avec le serveur ONELAB (client "interfacé").

L'interface ONELAB ne construit donc aucun méta-langage, ni format de fichier commun et le serveur ONELAB ne sait rien a priori des logiciels qu'il est amené à commander. Toute l'information à teneur technique mathématique ou physique vient des clients, ce qui assure que l'interface ONELAB se cantonne bien dans un rôle ancillaire d'interface et que l'on se prémunit du risque d'aboutir à une librairie tentaculaire et coûteuse à développer et maintenir sur le long terme. Les clients eux-mêmes sont les seules clés d'interprétation des données stockées sur le serveur ONELAB, c'est-à-dire que ONELAB ne peut les comprendre/interpréter qu'au travers de leur utilisation par les clients et en fonction des diagnostics et des résultats reçus.

Ceci constitue une différence notable par rapport à la plupart des approches classiques d'écriture d'interfaces (graphiques ou non) de pilotage de codes de calcul scientifique. Habituellement en effet, l'interface est écrite de manière *ad hoc* pour un solveur donné : en fonction des choix de l'utilisateur, l'interface génère le jeu de données nécessaires au code de calcul dans son format natif, puis exécute celui-ci. L'approche de ONELAB est différente en ceci que l'interface ignore la syntaxe propre des logiciels pilotés ; ce sont ces derniers qui interagissent avec l'interface et requièrent les données dont ils on besoin.

Couche experte

En pratique, un modèle numérique se compose de deux choses: un schéma de résolution paramétrique et un ensemble de valeurs pour les paramètres de ce schéma. Le modèle est opérationnel lorsque le schéma de résolution (la séquence des opérations à effectuer) et les valeurs des paramètres ont été déterminés de façon à ce que le modèle soit exempt d'erreur et fournisse les résultats désirés.

Partant d'un modèle opérationnel, la question se pose cependant rapidement à l'utilisateur de savoir dans quelle mesure on peut s'écarter des paramètres initiaux sans détériorer la qualité, voire la validité (physique), de la solution obtenue. Si, d'autre part les modifications de paramètres sont plus importantes, la question devient alors de savoir comment modifier le schéma de résolution pour s'adapter aux nouvelles caractéristiques du problème. La fonction de la couche experte est de fournir à l'utilisateur une aide à la détermination du schéma de résolution est des "bons" paramètres d'entrée pour chaque cas particulier de simulation. À cette

fin, la fonctionnalité d'intégration décrite précédemment (l'interface abstraite) n'apporte pas de solution et une fonctionnalité complémentaire doit être ajoutée.

D'une façon générale, le projet ONELAB n'aborde pas ce problème de façon théorique. Nous pensons que la définition *a priori* des conditions générales et théoriques de validité/qualité/ précision d'un modèle est illusoire et de surcroît contre-productive dans les cas d'applications pratiques. L'approche ONELAB est plus pragmatique, plus élégante, mais aussi plus générale. Outre le fait qu'elle permet l'utilisation de différents clients dans un environnement unique, l'environnement ONELAB est également doté d'un mécanisme d'accumulation des connaissances acquises au cours des simulations effectuées. Chaque simulation effectuée apporte en effet un certain nombre d'informations utiles (qu'elles soient positives ou négatives) concernant le problème étudié. Il est en règle générale du ressort exclusif de l'utilisateur d'archiver, traiter et hiérarchiser ces données de simulation. Aucun outil prévu à cette fin n'existe en pratique et, pour compenser l'absence de la couche experte "propriétaire" qui caractérise les plateformes commerciales, c'est précisément en fournissant un tel outil que ONELAB pourra reconstruire en quelque sorte "de l'extérieur" la composante "experte" qui est implémentée en dur "de l'intérieur" dans un logiciel commercial tel que COMSOL.

L'accumulation de connaissance se passe à deux niveaux. Tout d'abord dans la description de schémas de résolution (ce que nous appellerons les métamodèles) validés et d'autre part dans l'élaboration d'outils d'aide au choix des paramètres du métamodèle se basant sur l'exploitation des résultats stockés dans une base de données de simulation. La base de données de simulation et la librairie de métamodèles constituent conjointement la couche experte de l'environnement de modélisation ONELAB.

Délivrables

Les délivrables du projet ONELAB sont :

1. L'interface abstraite ONELAB vers les logiciels open source de calcul scientifique, sous forme :
    a. d'une librairie générique d'échange de données
        i. de CAO/maillage
        ii. de paramètres physiques, de contraintes, de paramètres de pilotage de code
        iii. de post-processing
    b. de interfaces nécessaires vers les logiciels clients en mécanique du solide, en mécanique des fluides et en électromagnétisme
2. Un ensemble représentatif de métamodèles couvrant les domaines de la mécanique du solide, de la mécanique des fluides et de l'électromagnétisme. Même s'ils peuvent aussi servir de cas-test, les exemples que l'on choisit doivent viser à mettre en évidence des simulations représentatives de simulations réelles. Un métamodèle est composé de deux éléments :
    a. une partie "simulation simple" qui représente la résolution de base du problème

étudié qui peut être exécuté pas à pas de façon interactive.

b. une partie “métamodèle” proprement dit qui utilise la partie “simulation simple” comme une boîte noire et peut servir par exemple à effectuer une optimisation complexe enchaînant différents calculs et utilisant différents modèles de complexité variable.

Ces exemples sont directement pilotables depuis l’interface ONELAB. Ils sont documentés pour permettre aux utilisateurs d’aller au-delà des “templates” s’ils le désirent.

Toutes les informations relatives au projet ONELAB sont disponibles sur le site internet http://onelab.info. Ce site contient, sous forme de wiki, la documentation des différents codes et modèles pilotés par l’interface ONELAB, ainsi que la plupart des informations contenues dans ce rapport.

Organisation du rapport

L’organisation du rapport suit celle du projet : les résultats obtenus pour les cinq tâches (T1 à T5) sont repris dans les cinq chapitres numérotés T1 à T5. Chaque chapitre est subdivisé en sous-sections correspondant au découpage en sous-tâches.

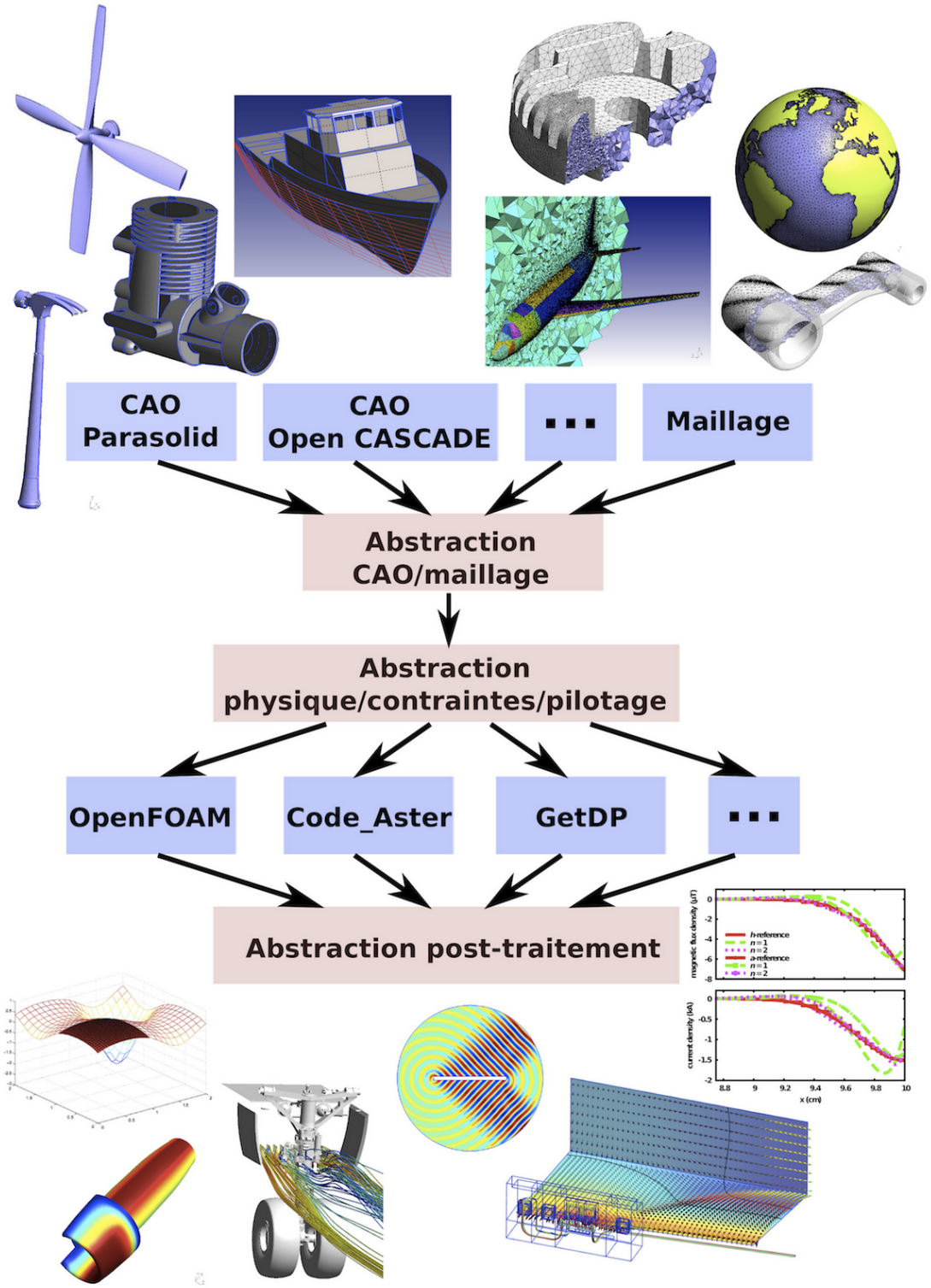

Figure 1. L'interface ONELAB est basée sur une triple abstraction : 1) de la CAO et du maillage, 2) des propriétés physiques, des contraintes et des paramètres de pilotage des codes open source "clients" et 3) du post-traitement.

T1 Spécification des interfaces

La première tâche du projet ONELAB est de concevoir l'interface abstraite vers les codes open source de calcul scientifique dans les domaines de la mécanique des fluides, de la mécanique du solide et de l'électromagnétisme. La spécification de cette interface se situe à trois niveaux : CAO/maillage (T1.1), paramètres physiques/contraintes/pilotage des codes (T1.2), et post-processing (T1.3).

T1.1 Spécification de l'interface CAO/maillage

Le but de cette tâche est de spécifier l'interface abstraite (API) vers la CAO et les maillages. Il a été décidé de baser cette interface abstraite sur l'interface de modèle géométrique "GModel" du code open source Gmsh développé par les membres du consortium.

La description géométrique pourra être fournie

- soit sous forme d'un maillage, auquel cas l'utilisateur pourra procéder à des opérations de nettoyage, de remaillage ou de simplification (pour mieux adapter la discrétisation à la physique et/ou à la méthode de calcul utilisée par le logiciel client) ou d'ajouts d'autres parties déjà discrétisées (afin de modifier un design existant) ;
- soit sous forme d'une CAO, en format natif tel Parasolid ou OpenCASCADE ou dans un format d'échange standard comme STEP ou IGES. Dans ce cas, l'utilisateur pourra également procéder à des opérations de nettoyage, de simplification ou d'ajout avant de procéder au maillage proprement dit.

L'abstraction vers la CAO et les procédures de (re)maillage est basée sur l'accès direct aux représentations géométriques des utilisateurs dans leur format natif, sans traduction de fichiers. A cet effet, le module géométrique de l'interface s'appuie sur une hiérarchie de classes abstraites permettant de représenter la topologie de n'importe quel modèle CAO, suivant le modèle B-REP ("Boundary REPresentation"). Ensuite, à chaque entité topologique abstraite est associée une implémentation concrète de sa géométrie pour chaque moteur CAO supporté (Parasolid, OpenCASCADE, Gmsh, etc.) Cette implémentation concrète se fait directement via l'API ("Application Programming Interface") de chaque moteur CAO. L'absence de traduction est un aspect crucial pour une utilisation industrielle, car toute traduction de modèles géométriques complexes est entachée d'erreurs liées aux tolérances géométriques et aux légères différences mathématiques entre les représentations natives et les représentations internes.

Après modifications éventuelles pour s'adapter aux exigences particulières du problème physique considéré ou de la méthode numérique utilisée par le logiciel client (ajouts de composants, raffinement ou simplification de maillage, etc.), l'utilisateur pourra mailler son modèle et exporter le maillage dans un format compris par les logiciels clients (MED, UNV, MSH). Tous les algorithmes de modification de modèles (ajouts, simplification, nettoyage et maillage), de même que les algorithmes de visualisation et de manipulation, seront écrits en termes des classes abstraites. Une nouvelle interface CAO ne demandera dès lors aucune modification à ces algorithmes—uniquement l'ajout d'une implémentation concrète

supplémentaire de la hiérarchie des classes d'accès.

Un point délicat au niveau des interfaces CAO dans un contexte industriel concerne la réparation de modèles CAO incorrects (présentant e.g. des chevauchements ou des trous non-physiques). Les développements récents des membres du consortium dans le domaine de la reparamétrisation des surfaces discrètes seront exploités dans ce cadre afin de proposer une nouvelle méthode de réparation de géométries défectueuses, indépendante du moteur CAO utilisé pour les créer.

Les modifications interactives ou scriptées de la géométrie seront gérées par la même interface abstraite de gestion des paramètres que celle conçue pour l'échange des propriétés physiques définies dans la section suivante.

T1.2 Spécification de l'interface physique / contraintes / pilotage

Le but de cette tâche est de spécifier l'interface abstraite vers les propriétés physiques, les contraintes et les paramètres de pilotage des différents logiciels clients.

La notion de métamodèle

On peut voir un modèle numérique standard (ou "simple") comme une application qui associe à une ensemble de variables d'entrée un ensemble de résultats numériques de sortie qui sont l'objet de la modélisation entreprise. Le modèle numérique simple est le plus souvent basé sur la séquence standard: maillage / pré-processing / résolution / post-processing et il est par définition explicite, en ce sens que les variables d'entrée sont données et suffisent pour obtenir les résultats souhaités. Ce sont des modèles "simples" de ce type que permettent de résoudre la majorité des suites logicielles commerciales de simulation.

Pour que le projet ONELAB soit attractif dans le monde industriel, il convient que, au-delà de l'aspect strictement économique associé à l'absence des coûts de licence, il offre sur le plan technique autant voire plus de possibilités que ses homologues commerciaux. Concrètement, il n'est pas rare en pratique qu'une partie des spécifications techniques imposées à un système concerne une ou plusieurs des données de sortie du modèle "simple". Le modèle contient dans ce cas une dimension implicite et est donc (partiellement) un problème inverse, ce qui invalide l'approche explicite "simple". De plus, puisque la plateforme autorise l'utilisation conjointe de logiciels différents, elle rend possible la comparaison de résultats obtenus avec des logiciels concurrents ou des approches théoriques différentes, offrant ainsi la possibilité d'un niveau de validation des résultats bien supérieur à ce qu'est en mesure d'offrir un code unique.

C'est au travers de la notion de **métamodèle** que l'environnement ONELAB entend donc dépasser les limitations inhérentes au modèle "simple". Concrètement, on attend d'un métamodèle qu'il offre les fonctionnalités suivantes:

1. une structure persistante permettant l'échange de paramètres entre codes différents (threads distincts du système), tournant sur la même machine ou sur des machines distantes,

2. la possibilité de définir des procédures de simulation (séquence de tâches numériques) séquentielles et éventuellement parallèles sans limitation de complexité,
3. une structure persistante permettant l'archivage et le traitement des résultats obtenus lors de chaque simulation, structure également évolutive et appelée à s'enrichir (tant par l'addition de données supplémentaires que par l'amélioration des procédures) au fil de l'expérience acquise,
4. la possibilité de travailler à un niveau d'analyse supérieur utilisant le modèle numérique simple (explicite) comme une boîte noire afin d'implémenter des schémas d'analyse et d'optimisation plus complexes, de relancer une simulation en tenant compte des résultats d'une première analyse, etc.
5. la possibilité d'utiliser des librairies scientifiques standard pour le traitement et le post-traitement direct (on the fly) des résultats de simulation,
6. assurer "de l'extérieur" la cohérence des modèles multi-code en encodant au niveau du serveur les relations de dépendance entre les paramètres mis en commun et les données d'entrée des différents clients,
7. outils de monitoring en cours de simulation,
8. outils de diagnostic et une aide à l'interprétation des erreurs.

Il est a noter que l'interface abstraite pourra également servir d'outil de validation des logiciels clients (et des (méta-)modèles associés) : en donnant des valeurs acceptables à certaines variables de sortie, on pourrait utiliser ONELAB (e.g. via un environnement de test comme CTest : www.cmake.org) pour effectuer des tests automatiques de validité des codes et des (méta-)modèles.

Suivent deux exemples de métamodèles:

Exemple 1: Dans un problème de magnétodynamique, la profondeur de pénétration fait partie des données de sortie, elle dépend de la fréquence de travail et des caractéristiques physiques des matériaux. Elle doit cependant être prise en considération lors de la définition des tailles de maille, qui elle est une donnée d'entrée. La règle de bonne pratique "au moins 3 éléments finis sur la profondeur de pénétration" est une spécification inverse en ce sens qu'elle nécessite la connaissance de la solution du problème. Dans l'environnement ONELAB, il est possible d'automatiser la réalisation d'une première simulation pour évaluer la profondeur de pénétration (pour laquelle aucune estimation analytique n'existe dans la cas non-linéaire) puis la prise en compte de cette valeur pour la réalisation d'une seconde simulation avec un maillage adapté. Avec des outils conventionnels, l'intervention de l'utilisateur serait nécessaire.

Exemple 2: Un métamodèle de poutre pourra combiner un simple modèle analytique en éléments de réduction, une modèle 2D et un modèle 3D, et encore éventuellement un modèle dual en contraintes. Ces différents modèles partagent une partie de leurs données d'entrées (dimensions, matériaux, etc). Chacun des modèles concurrents a ses avantages et inconvénients en termes de précision, temps de calcul, applicabilité, etc... On peu ainsi établir une forme de cartographie de l'espace des paramètres du modèle (y compris des caractéristiques de la discrétisation telles que une carte de taille, etc.) en terme des différentes modèles disponibles. Le métamodèle peut alors être augmenté d'une logique de sélection de modèle permettant son exploitation optimale en fonction des exigences d'exactitude, de temps et de précision donnée par l'utilisateur.

Abstraction des propriétés physiques

Nous sommes jusqu'à présent restés relativement vague quant à la définition exacte de ce que l'on entend par "paramètres du modèle". Le terme paramètre est d'ailleurs peut-être un peu réducteur en ce qu'il sous-entend dans le langage mathématique usuel une valeur spécifiquement numérique qui est laissée indéterminée lors de la définition d'un problème, que l'on qualifie alors de paramétrique.

Dans le cadre de la modélisation numérique, il est d'autres types de grandeurs que l'on peut souhaiter laisser indéterminées lors de la définition d'un modèle paramétrique. Il y en a essentiellement trois:

- des chaînes de caractères: nom de matériaux prédéfinis, nom de formulation, …
- des fonctions: le modèle paramétrique attend une relation algébrique entre 2 (ou plus) inconnues (p.ex. la relation constitutive d'un matériau) mais l'expression explicite de cette relation est laissée libre au niveau du modèle paramétrique et doit être fournie par l'utilisateur avant l'exécution du modèle,
- des régions abstraites: le modèle paramétrique travaille sur base de régions abstraites (p.ex. un domaine conducteur, une condition limite de type Neumann, …) et c'est à l'utilisateur d'identifier les zones du maillage avec ces entités abstraites.

L'abstraction de la définition des propriétés physiques, des contraintes et des paramètres de pilotage des logiciels clients sera également réalisée au travers de l'utilisation du serveur de paramètres permettant de stocker de manière unifiée toutes les informations requises par le modèle éléments finis sous-jacent. Chacun des types de paramètres considérés (nombres, chaînes de caractère, fonction, région) sera implémenté sous forme d'une classe C++ spécifique. Les propriétés physiques (matériaux) pourront être fournies par une loi de comportement linéaire ou non linéaire, une constante, un tenseur, une fonction analytique, des données émanant de mesures, etc. Les contraintes pourront être associées à des entités géométriques de dimensions diverses (points, lignes, surfaces, volumes) et imposées elles aussi au moyen de constantes, vecteurs, tenseurs, fonctions qui varient dans le temps, des résultats de mesures, etc.

Base de donnée de simulation

Comme expliqué plus haut, le métamodèle est doté d'une "mémoire" matérialisée par une base de données de simulation dans laquelle sont stockées les données d'entrée et de sortie de chacune des simulations effectuées. L'objectif est de faire en sorte que les informations obtenues aux étapes antérieures ne soient pas perdues mais restent utilisables (par un traitement adéquat des données accumulées dans la base de données) et interprétables afin de guider l'utilisateur vers un choix de paramètres valide correspondant à un modèle opérationnel. De cette façon, on enrichit au fil des essais et erreurs une heuristique propre à chaque problème et on reconstruit en quelque sorte "de l'extérieur" une partie importante de la fonctionnalité experte qui est implémentée en dur "de l'intérieur" dans un logiciel comme COMSOL.

Cette base de données de simulations permet également la comparaison des résultats obtenus, pour le même problème, avec des modèles concurrents. En ce sens elle contribue au

choix non seulement des "bonnes" variables d'entrée, mais également au choix de la "bonne" formulation, du "bon" logiciel, etc... Concrètement, en plus de l'espace des paramètres, les information suivantes sont archivées:

- des "diagnostics" des logiciels clients :
    - diagnostics de compatibilité (paramètre manquant, formulation incomplète, région géométrique non-définie, etc.)
    - diagnostics concernant la solution (convergence, nombre de pas de temps moyen, erreur, temps de calcul)
    - diagnostics "temps réel" (état d'avancement du calcul, mémoire utilisée, etc.)
- des solutions (au niveau du post-processing), sous forme
    - de "pointeurs" vers des fichiers (e.g. .pos, .gnuplot, etc.)
    - de valeurs numériques (e.g. grandeurs globales)
    - d'images

Implémentation

Initialement, nous pensions utiliser uniquement un langage interprété (LUA ou Python) pour concevoir cette interface bi-directionnelle entre le serveur et les logiciels clients. Afin d'être plus flexibles quant à l'utilisation de l'interface ONELAB sur différentes plateformes, nous avons décidé de modifier notre approche :

- Le coeur de l'interface vers les propriétés physiques, les contraintes et les paramètres de pilotage des codes sera réalisé sous forme d'une librairie C++ (le même langage utilisé pour l'interface CAO/maillage et l'interface de post-processing).
- Les parties spécifiques aux différents logiciels clients seront écrits soit en C++, soit en langage interprété (e.g. Python, Matlab), soit dans le langage natif du logiciel client.

T1.3 Spécification de l'interface post-processing

Le but de cette tâche est de spécifier l'interface abstraite (API) vers les données de post-traitement. Il a été décidé de baser l'approche sur l'interface actuelle de post-processing de Gmsh, et de l'étendre pour accéder aux données des logiciels clients :

- il a été décider de supporter le format d'échange de données MED3, développé par EDF et le CEA, qui est le format natif de Code_Aster (et de Code_Saturne) et sur lequel repose la plateforme Salomé ;
- il a été également décidé d'évaluer les capacités actuelles du format CGNS, en vue d'une possible intégration dans l'interface ONELAB.

T2 Interface CAO / maillage

T2.1 Implémentation de l'interface abstraite

L'interface abstraite vers la CAO et les maillages est basée sur le modèle géométrique GModel de Gmsh. L'implémentation est réalisée en C++, et repose sur une hiérarchie de classes abtraites, dérivées de la classe GEntity. Elle est accessible dans le répertoire `gmsh/Geo` du logiciel Gmsh. Diverses implémentations concrètes en lecture existent pour le noyau CAO Gmsh, OpenCASCADE, ParaSolid et ACIS. Pour l'écriture, la classe GModelFactory est en cours de développement pour le noyau CAO Gmsh et OpenCASCADE.

L'interface abstraite CAO/maillage est présentée en détail dans l'annexe B.

En conjonction avec cette interface abstraite vers les géométries et le maillage, Gmsh est devenu un client du serveur ONELAB de gestion des propriétés physiques, des contraintes et des paramètres des logiciels de simulation (voir chapitre suivant). Dans l'implémentation actuelle, cela signifie que Gmsh sert à la fois de serveur et de client ; et que des paramètres géométriques peuvent être modifiés en temps réel de la même manière que les paramètres des logiciels de simulation.

T2.2 Reparametrisation et nettoyage de géométries

Les fonctionnalités de reparamétrisation sont présentées dans l'annexe C.

T3 Interface physique / contraintes / pilotage

T3.1 Implémentation de l'interface abstraite

Organisation logique

Le métamodèle est la réponse concrète apportée aux spécification 2., 4. et 5. de la liste établie à la section T1.2. Pour satisfaire à ces spécifications, le métamodèle ONELAB s'organise en trois niveaux, à savoir, en allant du plus général au plus particulier:

- le métamodèle proprement dit (exactement un serveur de paramètres)
- la simulation (exactement un record dans la base de données de simulations)
- la tâche (exactement un client).

Une **tâche** est une étape de simulation qui est réalisée par un client bien défini. Une **simulation** est alors une séquence de tâches effectuée dans un ordre bien déterminé. Elle donne lieu à exactement un enregistrement dans la base de donnée de simulation. L'organisation en trois niveaux permet d'articuler de façon harmonieuse les aspects interactifs et non-interactifs que l'environnement ONELAB entend offrir à l'utilisateur.

Dans le cas d'une **utilisation interactive** de ONELAB (plutôt orientée vers les applications didactiques), les différentes tâches de la simulation sont exécutées pas à pas et la possibilité est offerte à chaque étape à l'utilisateur de modifier un ou plusieurs paramètres et de relancer la simulation. Vu que ONELAB garde une trace de la dépendance entre les clients et les paramètres, la simulation est reprise au niveau approprié pour que les paramètres modifiés soient effectivement pris en compte sous pour autant relancer systématiquement la simulation à partir du début et reproduire inutilement les parties de la simulation qui ne sont pas affectées.

La couche "métamodèle" proprement dite est quant à elle plutôt destinée à l'**utilisation non-interactive** (ou avec un niveau d'interactivité réduit, i.e. de type expert) de ONELAB. C'est à ce niveau que l'utilisateur peut "programmer" des schémas de résolutions plus élaborés (incluant des analyses paramétriques, des boucles d'optimisation, plusieurs modèles concurrents que l'on souhaite comparer, des couplages multi-physiques faisant intervenir plusieurs solveurs et des étapes de conversion de fichier de maillage ou de solution, etc.).

Communication par sockets

Au niveau informatique, le contrôle des logiciels s'effectue via une approche client-server. Les communications entre process s'effectuent soit sur socket TCP/IP, lorsque le client est distant, soit directement en mémoire lorsque le client et le serveurs sont compilés ensemble au sein d'une application monolithique).

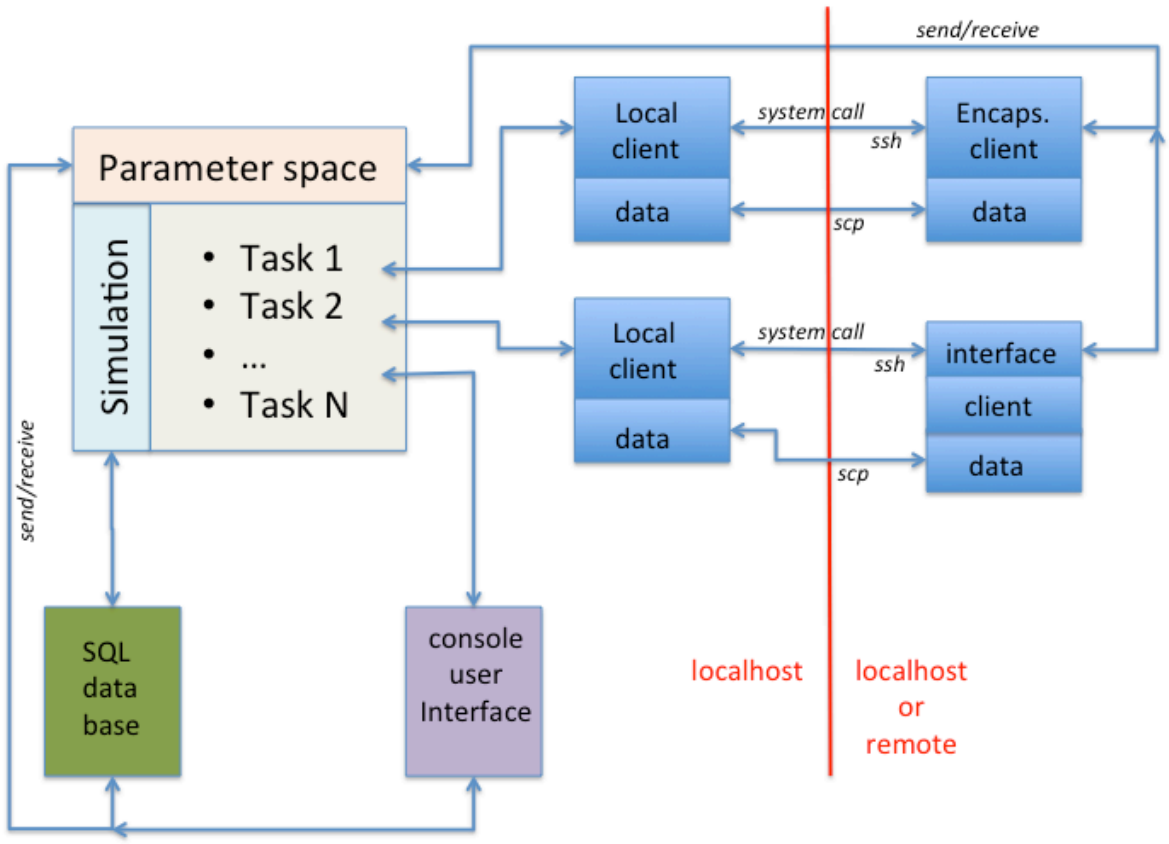

Figure 2. Représentation schématique de l'implémentation client-serveur de ONELAB avec communication par sockets. En haut à gauche le métamodèle avec son espace de paramètres unique et la liste des tâches décrivant les opérations successives de la modélisation. Chaque tâche communique avec un client local, qui, le cas échéant communique avec un client distant via un socket. La séparation physique entre programmes correspondant à des processes-système différents, ou tournant sur des machines différentes, est représentée par la ligne verticale rouge.

L'approche par socket a de nombreux avantages. Elle permet

- de créer automatiquement (et dynamiquement) une interface graphique permettant à l'utilisateur d'entrer ses données de manière interactive (par exemple, en sélectionnant à la souris les entités géométriques, des fonctions prédéfinies dans une base de donnée, etc.) ;
- de créer à la volée les jeux de données pour chaque logiciel client interfacé, ainsi que la ligne de commande pour lancer le calcul localement ou à distance avec les bons paramètres d'exécution.

Onelab.h

Pour être utilisable sur le plus grand nombre de plateformes informatiques, nous avons choisi d'implémenter la librairie en C++. La libraire peut-être soit utilisée de manière indépendante (connection aux codes de calcul via sockets), soit linkée "en dur" (iOS). Cette approche permettra d'intégrer l'interface ONELAB complète dans les plateformes (iOS) où imposer

l'utilisation d'un langage interprété n'est pas possible ou désirable.

Le header "onelab.h" contient le code C++ décrivant les structures de données du modèle client-serveur. La structure de serveur (classe `onelab::server`) contient une configuration de l'espace des paramètres (classe `onelab::parameterSpace`). Si la librairie est compilée en dur, les clients ONELAB communiquent directement avec le serveur ONELAB (un singleton). Si le serveur est indépendant, on utilise les mêmes appels clients/serveur, mais via un socket. Tous les clients dérivent de la classe abstraite `onelab::client`.

La spécification d'un paramètre (classe abstraite `onelab::parameter`) comprend:

- un type (variable, fonction, groupe)
- le nom des logiciels clients utilisant cette variable
- une aide courte (optionnel)
- une aide longue (optionnel)
- une valeur par défaut
- une valeur min/max (bornes) pour les valeurs numériques, ou valeurs acceptables pour entiers, etc.
- une catégorie, donnée sous forme d'arborescence (e.g. "Physical parameters/Magnetic Permeability" ou bien "Geometrical parameters/Width of airgap", ou encore "Advanced/ Solver parameters/Relaxation factor")
- un flag disant si le paramètre est obligatoire ou optionnel

Dans la représentation graphique de l'espace des paramètres, l'arborescence décrite dans la catégorie peut mener e.g. cacher certains paramètres avancés (e.g. si "Advanced" est le premier niveau dans l'arborescence). Les classes `onelab::number`, `onelab::string`, `onelab::region` et `onelab::function` dérivent toutes de la classe `onelab::parameter` et y ajoutent chacune leurs spécificités.

Le code source de l'interface abstraite est disponible dans `gmsh/Common/onelab.h`.

Interface graphique

Une première implémentation du serveur a été réalisée dans Gmsh, avec une interface graphique basée sur FLTK (http://www.fltk.org). Dans cas, Gmsh est à la fois un serveur et un client, ce qui permet d'agir de manière unifiée à la fois sur les paramètres de géométrie et sur les paramètres des solveurs.
L'interface graphique du serveur ONELAB dans Gmsh est implémentée dans `gmsh/Fltk/onelabWindow.{h,cpp}`.

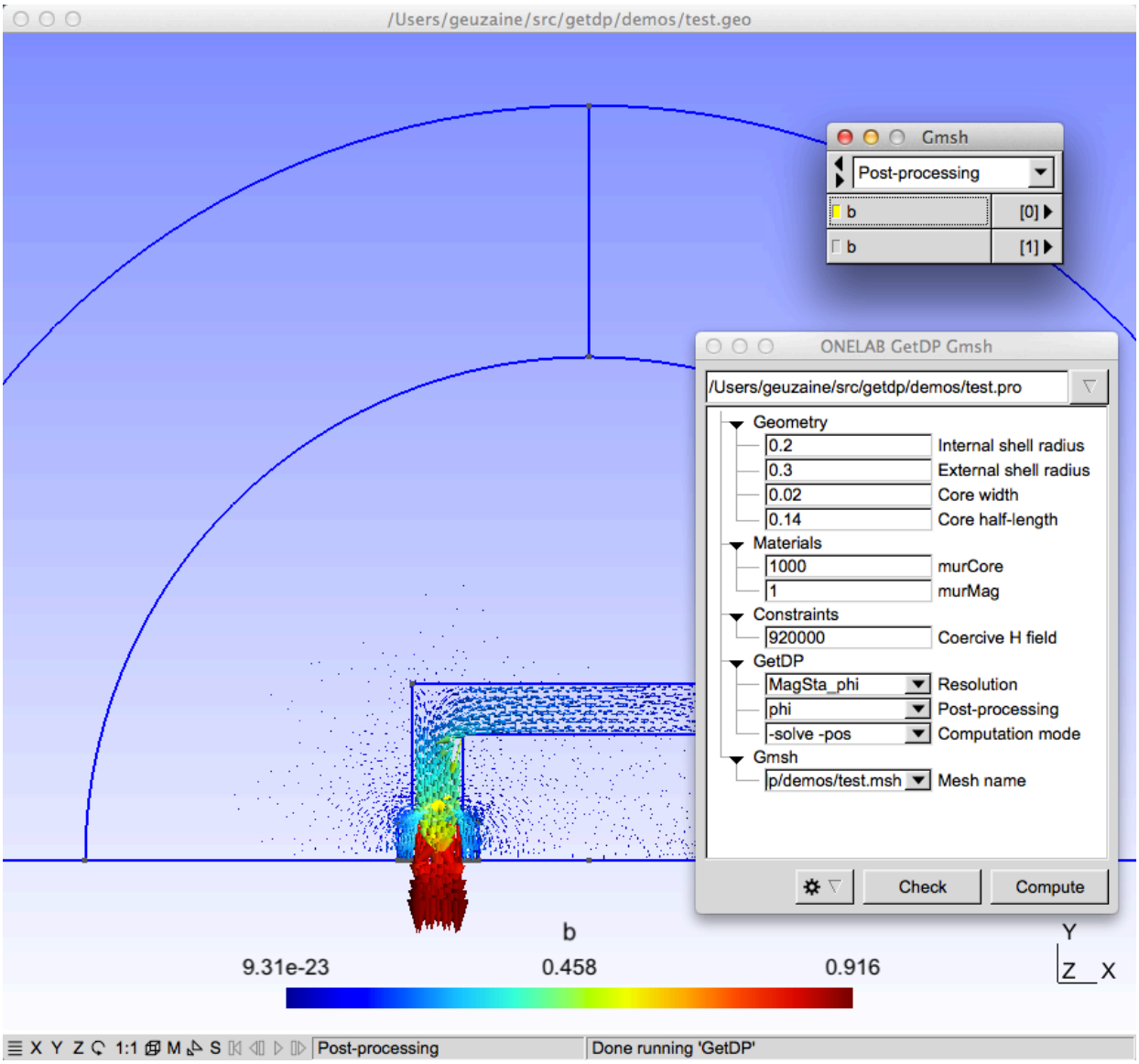

Figure 3. Interface graphique au-dessus de `onelab::server` dans Gmsh. Le serveur pilote ici deux clients : Gmsh et GetDP. Les paramètres "Internal shell radius" et "External shell radius" sont utilisés à la fois par Gmsh et GetDP.

T3.2 Interface vers OpenFOAM

On distingue plusieurs types de clients dans l'environnement ONELAB. Il y a d'abord deux clients "utilisateur" (l'un graphique, l'autre en mode console) permettant à l'utilisateur d'interagir directement sur le déroulement de la simulation au travers du serveur de paramètres. Il y a ensuite les clients solveurs dits "encapsulés". Il s'agit des solveurs in-house (gmsh, getdp, dg) pour lesquels le code source est disponible et la connaissance de celui-ci suffisante pour permettre l'implémentation en dur des interactions avec le serveur ONELAB.

Pour tous les autres solveurs, une syntaxe d'intefarçage doit être établie. Il est rapidement

apparu que cette syntaxe est pour une large part générique, c'est-à-dire identique pour tous les clients.

Syntaxe d'interfaçage

L'interfaçage ONELAB se fait en augmentant les fichiers d'entrée conventionnels du solveur d'un certain nombre de commandes destinées à être interprétée par le client ONELAB correspondant au solveur en question. Le fichier augmenté est le nom du fichier initial auquel on a ajouté l'extension caractéristique "_onelab". Ces commandes permettent de définir des paramètres et leurs attributs d'une part, et d'en récupérer la valeur telle que stockée sur le serveur d'autre part. L'interfaçage se fait au moyen de quatre commandes seulement, lesquelles sont maintenant détaillées.

1. Définition d'un paramètre et de ses attributs : "onelab.number"

Exemple:

```
onelab.number Tcold.Help(Applied cold temperature);
onelab.number Tcold.Default(77); Tcold.Min(50); Tcold.Max(100);
onelab.number NumStep.Help(Number of time steps); NumStep.Value(200);
onelab.number TimeStep.Help(Time step in seconds); TimeStep.Value(0.025);
```

On définit ici 3 paramètres (Tcold, NumStep et TimeStep), avec un certain nombre d'attributs.

2. Évaluation d'un paramètre: "onelab.getvalue()"

La commande `onelab.getvalue()` permet de récupérer la valeur d'un paramètre, telle qu'elle est stockée sur le serveur (et donc ayant potentiellement été modifiée par l'utilisateur ou un autre client) et de substituer cette valeur comme une chaîne de caractère dans le contexte original du fichier de donnée spécifique.

Exemple:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.7.1                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

nu              nu [ 0 2 -1 0 0 0 0 ] onelab.getValue(nu);


// ************************************************************************* //
```

3. Inclusion d'un fichier: "onelab.include"

Exemple:

```
onelab.include(filename)
```

4. Branchement conditionel booleen: "onelab.iftrue"

Exemple:

```
onelab.number FormulationA.Default(1);
onelab.iftrue(FormulationA)
  onelab.include(Formulation_a.pos)
onelab.else
onelab.include(Formulation_phi.pos)
onelab.endif
```

Comme on n'a pas de parser ONELAB, le branchement dépend d'une simple variable booléenne (ou entière). Le calcul de cette variable lorsqu'il n'est pas trivial et effectué au niveau du métamodèle.

L'interfaçage permet à ONELAB, ici encore, de rendre à l'utilisateur un certain nombre de services supplémentaires que le client solveur seul n'est pas en mesure de lui fournir. Il permet par exemple de centraliser la définition des paramètres qui, sinon, sont éparpillés dans plusieurs fichiers, voire plusieurs répertoires. De plus, les commandes d'include et de branchement conditionnels permettent de structurer les fichiers d'entrée du client de façon à ce qu'ils décrivent, non plus seulement le problème étudié mais au contraire toute une famille de problèmes. C'est l'étape d'interprétation par ONELAB (laquelle s'apparente à une étape de pré-compilation vis-à-vis d'un compilateur par exemple) qui est alors chargée de générer le fichier d'entrée unique en introduisant dans le fichier d'entrée symbolique les valeurs particulières obtenues du serveur.

T3.3 Interface vers Code_Aster

Le logiciel multi-physique ELMER () a été utilisé jusqu'à présent pour les simulations dans le domaine des structures. Son interfaçage a été réalisé sur le modèle expliqué à la section précédente. L'interfaçage effectif avec Code_Aster se fera effectué lors de la prochaine étape du projet.

T3.4 Interface vers GetDP

Le modèle abstrait du problème est déjà très structuré dans le logiciel GetDP : nous ne pensons pas qu'il soit dès lors utile/raisonnable de le redéfinir par ailleurs dans un "driver" externe.

Pour GetDP, nous avons donc décidé:

- d'exploiter directement les champs DefineConstant, DefineFunction et DefineGroup du langage .pro : en modifiant légèrement le parseur, GetDP échange désormais les paramètres définis dans DefineConstant avec le serveur ONELAB. DefineFunction et

DefineGroup sont encore à implémenter.

- A chaque lecture du fichier de donnée .pro GetDP interroge le seveur ONELAB, ce qui permet une gestion simple des erreurs, et une construction dynamique de l'espace des paramètres possibles (en fonction de “flags” donnés, certaines options deviennent accessibles ou non). Avec cette approche, gérer les dépendences entre paramètres dans la couche experte est inutile : c'est le modèle template .pro qui s'en charge. (Comme mentionné précédemment créer une couche abstraite qui garantit que le problème est bien posé est très ardu---cf. efforts du groupe de recherche du Prof. Kettunen de la Tampere Technical University dans ce domaine; cf. également Eficas, le “structurateur” de définition de problème utilisé par Code_Aster.)
- GetDP transmet également au seveur ONELAB une liste d'opérations possibles en fonction des choix ci-dessus (noms de "résolutions" et noms des opérations de post-pro disponibles)

Syntaxe provisoire dans le langage de GetDP :

- `DefineConstant[ CircuitCoupling ];`
- `DefineConstant[ CircuitCoupling = 0 ];`
- `DefineConstant[ CircuitCoupling = {0, Choices {0,1}, ShortHelp “Enable circuit coupling?”} ];`
- `DefineConstant[ CircuitCoupling = {0, Min 0, Max 1, Step 1} ];`
- `DefineConstant[ CircuitCoupling = {0, Help “Set to 1 to couple with electronic circuit?”} ];`

Cette syntaxe a également été intégrée dans le language de définition de géométrie (.geo) de Gmsh. Grâce à cette syntaxe commune, un métamodèle peut dès à présent être écrit dans les languages respectifs de GetDP et Gmsh, et inclure des fichiers de définition de paramètres communs.

T4 Interface post-traitement

L'interface abstraite de post-traitement de Gmsh est en cours d'extension pour supporter les nouveaux types de donnéées des logiciels clients (principalement MED3). Le post-traitement client/serveur sé́riel et parallèle est également en cours de développement.

Cette interface est présentée dans l'annexe C.

T5 Documentation et validation

Le travail de rédaction de la documentation vient de débuter. Il est réalisé de manière collaborative sous forme d'un wiki à l'adresse http://onelab.info.

T5.1 Documentation et validation de l'interface vers OpenFOAM

T5.2 Documentation et validation de l'interface vers Code_Aster

T5.3 Documentation et validation de l'interface vers GetDP

T5.4 Documentation générale