Code-coupling strategy for efficient development of computer software in multiscale and multiphysics nonlinear evolution problems in computational mechanics

Adnan Ibrahimbegovic¹, Rainer Niekamp², Christophe Kassiotis³, Damijan Markovic⁴, Hermann G. Matthies²

¹Ecole Normale Supérieure, Laboratory of Mechanics and Technology, Cachan, France, e-mail: ai@lmt.ens-cachan.fr
²TU-Braunschweig, Institute for Scientific Computing, Braunschweig, Germany
³Ecole des Ponts, Laboratory St. Venant, Paris, France
⁴EDF, SEPTEN, Lyon, France

1. Introduction

From its early days and focus upon the finite element method development (e.g. Zienkiewicz [1971], Bittnar [1998]), the computational mechanics has developed into a very broad area, much driven by various industrial applications (e.g. Oden et al. [2003]). Very diverse problems are presently of interest for research and expertise of computational mechanics community, such as biomechanics, nanotechnology, predictive home security etc. Typical formulations of interest are very much interdisciplinary, and ought to be formulated in terms of multiphysics problems: such as thermomechanical coupling, fluid-structure interaction, coupling of probability and mechanics. Moreover, in mathematical interpretation, the vast majority of these problems are highly nonlinear evolution problems. Thus, in trying to provide the most efficient solution procedure for any such problem, one is often prompted to exploit multiscale modeling and computations, which consider the most appropriate spatial and temporal evolution for each particular field. Providing the best theoretical formulation for any such problem is one of the currently most important challenges that has been addressed in a number of recent works on multiscale modeling and computations (e.g. Feyel and Chaboche [2002], Ibrahimbegovic and Markovic [2003], Hughes [2005], Ibrahimbegovic and Brank [2005], Ladeveze et al. [2005], Oden et al. [2005], Hautefeuille et al. [2012], among others).

However, the theoretical formulation of multiphysics and multiscale problems is not the only challenge. We argue here that providing the corresponding computational tools, or computer software, presents even a bigger challenge, with much more benefits for bringing this technology to bear upon the current engineering practice. Namely, as the problem complexity grows, many existing codes are found insufficient to meet the new requirement or even completely obsolete. Thus, a number of new developments have sprung in starting the development of new codes for multiphysics and multiscale problems, where higher level programming languages or existing libraries are used to accelerate the software development; case in point are: code in Java (e.g. Ehmerendy [2009]), Smalltalk (e.g. Dubois-Pelerin et al. [1992, 1998]), or Comsol (e.g. see Comsol [2010]). However, the software products of this kind are not necessarily the most efficient. In fact, although some tentative to increase the code
efficiency are made, many such codes are mostly used in academic research environment, very far from the industry applications.

Thus we propose here very different software development strategy, where the final software product is capable to fully integrate existing computer codes. The proposed strategy not only significantly accelerates the final code development, but even more importantly allows to directly include the existing codes that have been extensively tested previously. Last but not least, the proposed strategy can directly provide a very appealing interface to user who are used to a particular software product making them accept the new code release and other novelties much faster.

We note in passing that the same strategies of coupling of existing codes are currently used by interpreters, such as Python (e.g. see Langtangen [2008], Gendre et al. [2009]) or Matlab (e.g. see Bindel [2011]). However, contrary to such an approach which sacrifices the efficiency in favor of flexibility while coupling typically the executable versions of existing codes, we target here the code coupling that is done at the compilation and linking time, and thus provide truly the single code as the final software product. Such a code, perhaps needless to say, will be both more robust and more efficient than any other code-coupling alternative.

Last but not least, in seeking the optimal results we show in this paper the need to focus upon the complete treatment of the present problem, and not only the code-coupling. More precisely, the proposed strategy ought to reconsider the theoretical formulation of multiphysics and multiscale problems, the optimal discrete approximations both in space and in time, and finally the most efficient software development based upon the code-coupling strategy. This is illustrated upon two model problems. The first one concerns the multiscale analysis of inelastic behavior of heterogeneous materials (e.g. see Ibrahimbegovic and Markovic [2003], or Markovic et al. [2005]), where the same code FEAP (e.g. Zienkiewicz and Taylor [2005]) is used at both scales. The second pertains to fluid-structure interaction, where FEAP is coupled with OpenFOAM (e.g. see OpenFOAM [2006]) in order to produce the final software product. Both chosen model problems allow us to illustrate the important concept of interface and the use of localized Lagrange multipliers (e.g. Park et al. [2005], Markovic et al. [2009]). In the multiscale problem the interface between the microscale and macroscale is defined in terms of macroelement boundaries. In the multiphysics problem the interface is the wetted structure boundary. The key role in defining the interface very clearly is played in choosing the original interpretation of theoretical formulation for multiscale strategy and fluid-structure interaction. The second important point of our strategy of using the localized Lagrange multiplier pertains to the optimal choice of discrete approximation for each sub-problem. This is illustrated in the case of multiphysics problem of fluid-structure interaction, where the finite element method is used for structure and finite volume method is used for fluids in order to construct optimal discrete approximations. The seamless connection of these two approximations is again achieved by the localized Lagrange multipliers and their corresponding interpolations in terms of radial approximations.
The outline of the paper is as follows. In the next section, we present the theoretical formulations of the multiscale problem for inelastic behavior of heterogeneous materials, along with the multiphysics problem of fluid-structure interaction. The discrete approximation of the latter problem is also presented in detail in Section 2, along with the corresponding proofs of convergence of sequential solution procedure and the illustrative examples showing what kind of results can readily be obtained in this manner. Section 3 provides a number of pertinent remarks on numerical implementation and the essence of proposed code-coupling procedure based upon the CTL-component template library tool (e.g. see Niekamp [2003], Niekamp et al [2009]). Concluding remarks are given in Section 4.

2. Theoretical formulation and discrete approximation of multiscale and multiphysics nonlinear evolution problems with interface

2.1. Localized Lagrange multipliers

In this section we first introduce the notion of interface and corresponding localized Lagrange multipliers, which allows reduction of the problem complexity. This is done in the simplest possible setting of domain decomposition where two domains are connected at the single point (see Figure 1).

![Figure 1.- Partitioned solution by domain decomposition where two domains are connected through a single point.](image)

The classical formulation of partitioned problem introduces the Lagrange multiplier imposing the equalities of field values in points 1 and 2, which are now separated by the partition. This leads to a particular constraint:

\[ c^{(12)} = u^{(1)}_\Gamma - u^{(2)}_\Gamma = 0 \]  

(1)

In the solution method, this kind of constraint is handled by the Lagrange multiplier \( \lambda^{(12)} \), and the resulting set of equations can be written as:
where the operator $A(.)$ represents the response of the system corresponding to system excitation $f^{(\alpha)}$. We can see from (2) above that the two systems are ‘visible’ to each other through the global Lagrange multiplier, physically representing the conjugate action to difference in field values in two subsystems. The main disadvantage of such a formulation is in keeping the partitioned systems tightly coupled, which thus leads to questionable gains of partitioned strategy.

However, an alternative to such a formulation is the use of localized Lagrange multipliers (e.g. Park et al. [2005]), which can introduce the notion of interface to separate the system into subsystems that will no longer be visible to each other. Namely, for the case illustrated in Figure 1 where the system partitioning concerns only one point, we will introduce an interface point that belongs to neither system. If the value of the field at that point is denoted as $u_f$, we can then rewrite the constraint in (1) as follows:

$$
c^{(1)} = u^{(1)}_{f} - u_f = 0
$$

$$
c^{(2)} = u^{(2)}_{f} - u_f = 0
$$

The system in (2) can also be rewritten by replacing the last term with a modified constraint equation:

$$
\delta \Pi_{total} = \delta \Pi^{(1)} + \delta \Pi^{(2)} + \delta \pi_{localized}
$$

$$
\pi_{localized} = \lambda^{(1)}T (u^{(1)}_{f} - u_f) + \lambda^{(2)}T (u^{(2)}_{f} - u_f)
$$

We note that the interface degrees of freedom in this case will separate two subsystems, and would not allow either to see the other one. The advantage of the localized Lagrange multipliers, here $\lambda^{(1)}$ and $\lambda^{(2)}$ is that they pertain only to a single sub-system. Thus, they can be handled in purely local computations, introducing the most appropriate scaling to improve the system conditioning etc.

2.2. Multiscale evolution problem

We could extrapolate these concepts directly towards the multiscale analysis of nonlinear evolution problem. In order to illustrate these ideas we refer to Figure 2, showing the macro and micro scale representation of 3-point bending test (see Ibrahimbegovic and Markovic [2003]).
The macro scale represents the structural level, with the discrete model constructed with what looks like the standard finite element mesh (‘black nodes’, with displacement denoted as $d^M$).

![Macro and micro model for 3-point bending test: black nodes denote the ‘macro’ scale and ‘white’ nodes the ‘micro’ scale nodal displacement; indicated possibilities to constructed structured finite element mesh, or nonstructured discrete element mesh](image)

Figure 2. - Macro and micro model for 3-point bending test: black nods denote the ‘macro’ scale and ‘white’ nods the ‘micro’ scale nodal displacement; indicated possibilities to constructed structured finite element mesh, or nonstructured discrete element mesh

However, there is another mesh of ‘micro’ nodes that can be placed inside each macro element. As illustrated in Figure 2, this mesh can be constructed not only by finite elements (e.g. see Ibrahimbegovic and Markovic [2003]), but also discrete elements (e.g. see Ibrahimbegovic and Delaplace [2003]). The total energy of such model can be decomposed in terms of macro energy, micro energy and the interface condition concerning the macro and micro interface boundaries $\Gamma^{Mm}$.

$$
\Pi(u^M, u^m, \lambda, \zeta_k) = \Pi^M + \Pi^m + \Pi_{\Gamma^{Mm}};
$$

(5)

The macro energy contains the contribution of the external nodal loads, defined as:

$$
\Pi^M = - \int_{\Omega^M} u^M \cdot b^M dV - \int_{\Gamma^M} u^M \cdot \tau dA
$$

(6)

The micro energy is the term containing the corresponding interpretation of inelastic mechanism at the origin of nonlinear evolution process. Denoting the latter symbolically by a set of (internal) variable $\zeta_k$, we can write the micro energy contribution in terms of the corresponding potential (we note that we can go to the finest possible scales at present in order to provide the best interpretation of internal variables; e.g. see Marenic et al. [2012]):

$$
\Pi^m = \int_{\Omega^m} \Psi^m(u^m, \zeta_k) dV
$$

(7)

The interface term can be defined according to:

$$
\Pi_{\Gamma^{Mm}} = \int_{\Gamma^{Mm}} \{ \lambda^{(1)}(x) \cdot [u^{(1)}_J(x) - u_J(x)] + \lambda^{(2)}(x) \cdot [u^{(2)}_J(x) - u_J(x)] \} d\Gamma
$$

(8)

where $\lambda^M = \lambda^{(1)}$, $\lambda^m = \lambda^{(2)}$ etc. We will further use the standard finite element interpolations for macro and micro displacement field:
By finally using the Dirac delta function for micro scale localized multipliers centered on each micro node on the interface, we can obtain from (8) a set of algebraic equations connecting directly the micro nodal displacement at the macro element interface with the macro element nodal displacements (see Figure 3):

\[
\left. u_{m+1}^{E} \right|_{\Gamma_{m,E}} (x^m) = \sum_{a \in \Gamma_{M,E}} N_{a}^{M,E} (x^m) d_{n+1}^{M,E} ; \quad \left. u_{m+1}^{\omega} \right|_{\Omega^m,\epsilon} (x^m) = \sum_{\alpha=1}^{n_\Omega} N_{a}^{m,\epsilon} (x^m) d_{n+1}^{m,\epsilon} ; \quad \forall \Omega^m,\epsilon \subset \Omega^M,\epsilon
\]  

(9)

Figure 3. – Interface between macro and micro scale in proposed multiscale model

On the other hand, by choosing the interpolation for localized Lagrange multipliers at the macro scale constructed by Dirac delta centered upon macro nodes, we simply obtain the standard finite element assembly procedure summing up the contributions from all macro elements attached to a particular macro node. The flowchart of the complete solution procedure can be written as:

\[
\bar{d}_{n+1}^{M,E} - \sum_{b} N_{b}^{M,E} (x_a) \bar{d}_{b,n+1}^{E} = 0 ; \quad \forall a
\]

\[
\Rightarrow \left. \bar{d}_{n+1}^{M,E} \right|_{\Gamma_{M,E}} = T^{E} d_{n+1}^{M,E}
\]  

(10)
where denotes the standard finite element assembly operator. The validation of our approach is provided first with respect to multiscale numerical simulation of 3-point bending test (see Figure 4).

Figure 4.- Three point bending test, solution procedure and quadratic convergence rate in typical steps for Newton method used both at macro and micro scales

We also illustrated in Figure 4 the quadratic convergence rate that one can achieve at both macro and micro scale by using the Newton iterative method. It is also possible to employ other iterative methods (e.g. Jacobi, Gauss-Seidel …), but the convergence is not always guaranteed, especially for the stiff systems (e.g. Niekamp et al. [2009]).

2.3. Multiphysics nonlinear evolution problem of fluid-structure interaction
In this section we further extend the proposed approach to multiphysics problem, and in particular fluid-structure interaction. The key idea that is exploited in order to define the interface pertains to ALE formulation (e.g. see Hughes et al. [1981], Donea et al. [2003]). Namely, on one side we define the structural equations of motion in Lagrangian formulation (where domain $\Omega_s$ follows the structure motion):

\begin{equation}
\mathcal{G}_s := \int_{\Omega_s} \rho_s \partial_t^2 \mathbf{u}^h \cdot \delta \mathbf{u}^h + \int_{\Omega_s} \sigma^h : \nabla \delta \mathbf{u}^h - \int_{\Omega_s} \mathbf{B} \cdot \delta \mathbf{u}^h - \int_{\partial \Omega_s} \lambda \cdot \delta \mathbf{u}^h = 0
\end{equation}

The index ‘$h$’ on each field indicates the discrete approximation, which is here constructed by using the finite element interpolations (e.g. Zienkiewicz and Taylor [2005]). On the other hand, the fluid equations of motion are set in Eulerian formulation (where domain $\Omega_f$ remains fixed in space):

\begin{equation}
\mathcal{G}_f := \int_{\Omega_f} \left( \mathbf{u}^h - \mathbf{u}^h_m \right) \cdot \delta \mathbf{u}^h + \int_{\Omega_f} \partial_t \mathbf{v}^h \cdot \delta \mathbf{v}^h + \int_{\Omega_f} \left( \mathbf{v}^h - \mathbf{v}^h_m \right) \otimes \mathbf{v}^h \cdot \delta \mathbf{v}^h - \int_{\Omega_f} \nabla \cdot \mathbf{v}^h D(\mathbf{v}^h) \cdot \delta \mathbf{v}^h + \left( \int_{\Omega_f} \frac{1}{\rho_f} \nabla \rho^h \cdot \delta \mathbf{v}^h + \int_{\Omega_f} \frac{1}{\rho_f^f} \nabla \mathbf{v}^h \delta \rho^h \right) + \mathbf{C}_L = 0
\end{equation}

The index ‘$h$’ on each field also indicates the discrete approximation, but this time constructed by using the finite volume interpolations (e.g. see Eymard et al. [2000]). The corresponding forms of structure and fluid equations of motion are obtained in agreement with choice of Lagrange multipliers. Namely, the Lagrange multiplier field on structure side in (12) is interpolated in agreement with the particular choice of finite element mesh, representing physically the corresponding pressure loading induced by fluids. On the other hand, the Lagrange multipliers on the fluid side are chosen with the Dirac delta approximations centered on the finite element nodes, which imposes the fluid mesh motion in strict agreement of the structure. Moreover, in the dynamic setting considered herein, the interface conditions ought to impose the continuity not only of displacement but also of velocities:

\begin{align*}
\begin{cases}
\mathbf{u} = \mathbf{u}_S & \text{continuity of displacements} \\
\mathbf{v} = \mathbf{v}_S & \text{continuity of velocities}
\end{cases}
\end{align*}

As illustrated in Figure 5, the constraints of this kind concern both the space and time approximations. The general development proposed recently (see Kassiotis et al. [2011a,b]) allow the most general conditions for space approximations connecting the finite element approximations for structure and finite volume approximations for fluid. These two approximations do not share the same parameters (nodal values for finite elements versus cell centers for finite volumes), and have to be connected by using the interface approximations in terms of radial approximations (see Kassiotis [2011a,b] for details).
Similarly, we can allow two different time steps to be used for structure on one side versus fluid on another (see Figure 5 for illustration). Yet, one can still provide the stability proof of the corresponding iterative scheme for geometrically nonlinear setting (see Kassiotis et al. [2011a,b]) for details); the latter can be written as:

\[
\left\| M_s^{\text{T}} D_s \left[ D_s M_s^{-1} \left( 1 - B_f (B_f^T M_f^{-1} B_f)^{-1} B_f^T M_f^{-1} \right) D_f^T \right]^{-1} D_s \right\| \leq 1
\]

(15)

where \( M_s \) is structure mass, \( M_f \) is fluid mass, whereas \( D_s \) and \( D_f \) depend, respectively, upon the structure and fluid field transfer towards interface. We note that the stability condition in (15) is rendered more stringent in the presence of the incompressibility constraint \( B_f \).

The numerical results for validation of the proposed solution procedure for fluid-structure interaction problems with increasing model complexity are presented in Figures 6 and 7. The former represents the well-known 1D benchmark of vortex-excited oscillation, which is compared against the experimental results (Anagnostopoulos, Bearman [1992]) and fully coupled solution procedure for fluid-structure interaction (Dettmar, Peric [2007]).

Figure 5.- Fluid-structure interaction problem: space and time continuity constraints

Figure 6.- 1D test example of vortex-excited disk oscillation: left) problem definition, right) result obtained by partitioned scheme, versus those in [7] Dettmar, Peric and [8] Anagnostopoulos, Bearman.
The second problem is the analysis of 3D case concerning the vortex shedding induced vibrations of the thick beam structure placed in lateral fluid flow. The illustrative snap shots of flow streamlines and pressure contours are presented in Figure 7.

*Figure 7.- Fluid-structure interaction of thick beam placed in lateral flow: left) flow streamlines, right) pressure contours.*

3. Software production issues and code-coupling with CTL

For the realization of the micro-macro-coupling the FORTRAN-code FEAP was taken both at micro and at macro-scale, where for the FSI-coupling FEAP, see e.g. FEAP [2011], (via the coFeap-component) was coupled with the C++-code OpenFoam, see e.g. OpenFOAM [2006] (via the oFoam-component). This was done using the component based software engineering. As the underlying middleware the Component Template Library (CTL) (Niekamp [2009], CTL [2005]) is used. Analogously to the localized Lagrange multiplier technique separating the physical components and decoupling the resulting systems of equations from the physical interface, the component based software engineering is formulated with independent software components whose functionality is hidden under software interfaces.

The CTL is a C++ template library, which uses template meta-programming to hide as much as possible technical details of unfamiliar components from the programmer. It has been successfully applied in computational applications like multiphysics simulation (Birken et al. [2009, 2010], Matthies et al. [2006], Srisupattarawanit et al. [2006], Kassiotis et al. [2011a,b]), multiscale simulation (Ibrahimbegovic, Markovic [2003], Markovic et al. [2005], Niekamp et al. [2009]), stochastic finite element analysis (Krosche, Hautefeuille [2007], Krosche, Matthies [2008]) and optimization (Niekamp et al. [2003]).

The CTL provides an Interface Definition Language (IDL), which is embedded in C++ by using template meta-programming techniques (Vandevoorde, Joussitis [2003], Alexandrescu [2001]) for generative programming. The CTL is invisible, where other frameworks use external programs to generate so-called stubs and skeletons (Corba [2006]). Instead, the CTL provides a concept called connect, which supports the binding of code to an interface. The overhead of making a call to a component is thereby minimal, implemented by the cost of a single virtual function-call. It allows seamless use of a large number of communication concepts, which
implies that neither the code of the component nor the code of the used application would need to be changed.

- shared libraries (UNIX)/dynamic-link library (Windows): Components are loaded at runtime into the process. Calling a component is always blocking, i.e. the calling component can not continue until the called one has completed.

- threaded shared library: Components are loaded at runtime into the process and run in a separate thread. Calling a component can be blocking or non-blocking.

- socket daemon (TCP): Components are running as an operating-system daemon listening on a predefined UNIX-port.

- Just-in-time daemon (TCP/SSH): Components are instantiated locally or remotely just-in-time of their use.

- static linkage: Early binding of normal C++-objects to Interfaces, providing interface polymorphy in C++

- Message passing (MPI/PVM)

The binding is performed just by a call to template function: `ctl::connect<IFace, Impl, Detail>()` connecting an interface (IFace) to an implementation (Impl) by using an optional specification of details (Detail) of this connection. This amounts to possibilities and flexibility that is unique for a component framework.
Figure 8 illustrates on the left side the dependence graph during classical (monolithic) linkage creating the standard version of FEAP, and on the right side component linkage creating a client communicating with coFeap component. Here for each method with name ‘XXX’ declared in simu.ci, its FEAP-based implementation can be found in simu_XXX_impl.f. The coFeap component (built as shown in the central part in Figure 8) is used for the structural part in fluid-structure interaction problem, as well as for the microstructure computation in the multiscale problem. Here we want to mention, that it was a job of one long day for Bob Taylor and one of the authors of this paper, to produce the prototype of the coFeap component.

While in the monolithic linkage case the application depends directly on the used library (Feap7.5.a in Figure 8), in the CTL-based linkage we can only see the interface (simu.ci). Consequently, the proposed approach provides two important advantages:

- The application does not have again to be recreated, if the module is modified.
- At run-time the application can decide, which components and how many instances of these components on which hardware is to be used.

Furthermore, due to the fact that the macroscale elements are perfectly decoupled by the localized Lagrange multipliers, in the multiscale computations we have optimal parallelization capabilities. More precisely, we can just start as many coFeap instances as CPU-cores that are simultaneously available by the –l tcp linkage. Also on the machine where the macro-feap processes we can add reasonably (number of cores -1) many micro-instances by FEAP using the asynchrony –l thread linkage.
Building the coFoam-component from OpenFOAM leads to a very similar dependency structure as given in Figure 8. In the fluid-structure interaction problem, the computational cost is strongly dominated by the fluid part. Therefore the best parallel efficiency can be yield by using the nested parallelization with parallelized implementation of OpenFOAM (see Kassiotis et al. [2011a,b]).

3.1 FORTRAN and Ansi-C-Bindings of CTL

When the CTL is used with the programming language C++, CTL-interfaces are just C++-classes and can be used as classes. For example, in the fluid-structure interaction problem, both the control and client-code are written in C++. For fluid part, this is OpenFOAM code. However, the code fragment invoking the solver of the structural part, implemented in FORTRAN for FEAP code, is given by:

```fortran
simu Solid("inputfile");
...
// set Neumann boundary
Solid.set_load(f[solid]);

Solid.solve();

// copy result into u[solid]
Solid.get_state(u[solid]);
```

Only on the server side a CTL-interface has to be connected to an implementation. For this purpose a connecting C++-source file, here for binding the OpenFOAM simulation code, is needed:

```cpp
connectCoFoam.cpp:

#define CTL_Connect
#include <cfdsimu.ci>
#include <ofoam.hpp>
...
void ctl_connect()
{ ctl::connect<CFDSimuCI, Ofoam<>, detail>(); }
```

This file has to be compiled and linked against the CTL-library (a dynamic shared object) and the library containing the needed implementations declared by the interface (in this case with the OpenFOAM libraries).

The CTL provides also language bindings for Ansi-C and FORTRAN, e.g. it is possible to transform an existing FORTRAN library, e.g. FEAP.a, into a remote accessible software component and to use a CTL component by FORTRAN applications. Especially into FORTRAN only a subset of the C++ features can be mapped. Therefore only a restricted set of interfaces can be used or implemented by FORTRAN and Ansi-C. The particular restrictions are:

- only CI classes, no CI libraries,
• methods returning and accepting argument types listed in the following subsection. These restrictions are not essential and do not prevent an efficient implementation.

### 3.1.1 Parameter Conversion CTL $\rightarrow$ FORTAN/Ansi-C

In the standard FORTRAN programming language no structured types like string or array (=vector) of a type are expressible. On the other hand such structures are needed to implement a type safe message passing. Therefore the interface is defined in such structures like an array which aggregates the information about number, type of data and the data itself to be transmitted between different processes. Hence, a translation of these structures to fundamental FORTRAN data types is needed.

#### Translation to Fortran/Ansi-C- Types:

<table>
<thead>
<tr>
<th>interface type</th>
<th>(\rightarrow)</th>
<th>FORTRAN representation</th>
<th>Ansi-C representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fundamentals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int1</td>
<td>(\rightarrow)</td>
<td>character</td>
<td>char</td>
</tr>
<tr>
<td>int2</td>
<td>(\rightarrow)</td>
<td>integer*2</td>
<td>short</td>
</tr>
<tr>
<td>int4</td>
<td>(\rightarrow)</td>
<td>integer*4</td>
<td>long</td>
</tr>
<tr>
<td>int8</td>
<td>(\rightarrow)</td>
<td>integer*8</td>
<td>long long(^*)</td>
</tr>
<tr>
<td>real4</td>
<td>(\rightarrow)</td>
<td>real*4</td>
<td>float</td>
</tr>
<tr>
<td>real8</td>
<td>(\rightarrow)</td>
<td>real*8</td>
<td>double</td>
</tr>
<tr>
<td>structures</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>(\rightarrow)</td>
<td>integer*8, character</td>
<td>long long, char</td>
</tr>
<tr>
<td>array&lt;fund.&gt;</td>
<td>(\rightarrow)</td>
<td>integer*8, fund.</td>
<td>long long, fund.</td>
</tr>
</tbody>
</table>

| *compiler dependend, e.g. gnu-C: long long, VisualC: int64 |

Any const modifier in the interface will be ignored. For example the interface signature

\[
\text{int1/}^{\star}\text{ret}/ \text{simu::solve (const real8/}^{*}\text{accuracy/}, \text{array<real8> /}^{*}\text{x/)}\]

corresponds to the FORTRAN subroutine

\[
\text{subroutine simu_solve_impl (accuracy, size_x, x, ret )}\\
\text{real8 accuracy}\\
\text{integer8 size_x}
\]
real*8 x(size_x)
character ret

and to the C-function

void simu_solve_impl(double* accuracy, long long* size_x double* x, char* ret )

The last argument ret corresponds to the return value specified in the interface signature.

The name simu_solve_impl is the composition of <CI_Name>_Impl.

A constructor defined in the interface must be implemented by a subroutine named <CI_Name>_new_CtorID_impl.

3.1.2 Creation of FORTRAN and Ansi-C-services

For the microscale problems and the structural part of the fluid-structure interaction problem, FEAP has to transformed into the coFeap component. This is done by using the following:

connectcofeap.cpp:

// FORTRAN binding
#define CTL_ConnectF

// FEAP is not multi-thread save
#define CTL_unSaveMT

// implementation function names start with simu_
#define CTL_ClassPrefix simu // or C-Binding
#include <simu.ci>

void ctl_connect() // bind to external implementation
{ctl::connect<simuCI, ctl::Extern::simuCI> ;}

(In the case of C-binding #define CTL_ConnectF must be replaced by #define CTL_ConnectC.)

In this case the implementation comes just via linkage (here of the simu_XXX_impl.o’s together with the FEAP library) into the component. As an example we consider for the set_load method.

simu.ci contains this method definition needed for control of the Neumann boundaries:

#define CTL_Method6 void, set_load, (const array<double> /*load*/), 1

The corresponding implementation is given in simu_get_load_impl.f according to:

subroutine simu_set_load_impl(loadsize, loadval)
implicite none
[...]
include ‘control.h’
integer*8 loadsize
real*8 loadval(loadsize)
integer i,j,k

k = 0
[...]
doi = 0,neq-1
j = mr(np(31)+i+k)
do while(j.lt.0)
k = k + 1
j = mr(np(31)+i+k)
enddo
hr(np(27)+i+k) = loadval(i+1)
enddo

3.1.3 Creation of FORTRAN and Ansi-C-clients

If a FORTRAN or C-program wants to use the CTL-interface simu.ci a C++source like clientF.cpp/clientC.cpp must exist.

clientF.cpp/clientC.cpp:

// either FORTRAN binding
#define CTL_ClientF
// or  C-Binding
//#define CTL_ClientC
#include <simu.ci>

This must be compiled and linked to FORTRAN/C-client. Now the subroutines with the names simu_XXX are in the client available.

3.1.4. Compiler/platform dependent configuration of CTL

For the successful linking of FORTRAN-code against the CTL-library on Unix- or Linux-systems the following compiler and linker flags must be set. The -fPIC (Position Independent Code) is only needed on 64bit platforms.

ifort Intel-compiler:

- CPPFLAGS =-fPIC -DCTL_FortranPostFix= -DCTL_FortranPostFix=_
- FFLAGS =-fPIC -names as_is
- LDLIBS =-lifcore

gfortran Gnu-compiler:

- CPPFLAGS =-fPIC -DCTL_FortranPostFix=_ -DCTL_FortranPostFix=_
3.2. Hierarchical ordering of simulation interfaces

Since version 1.3, the CTL allows an interface to extend other interfaces. This enables to organize the different facilities of different simulation codes in a hierarchy of interfaces. If a code implements an interface, it automatically implements all interfaces directly or indirectly extended. Therefore this ordering can be used to optimize the interchangeability of simulation codes, e.g. in multi-physical simulation.

Example Dirichlet-Neumann coupling: (instead of CTL-interface syntax a more readable IDL is used.)

Any useful simulation code should at least implement an interface `stationarySimulationCI` like:

```cpp
interface stationarySimulationCI
{
    stationarySimulationCI (const string /*input-file*/);
    void solve();
    void get_state(array<real8>/*state*/) const;
};
```

Using (for a stationary problem) Gauss-Seidel- or Jacobi-type solution schemes each Neumann controlled part of a coupled simulation needs at least to implement the interface:

```cpp
interface stationaryNeumannCI: extends stationarySimulationCI
{
    void set_load(const array<real8>/*load*/);
};
```

In this case the part controlled by Dirichlet-conditions needs at least to implement an interface `stationaryDirichletCI` like:

```cpp
interface stationaryDirichletCI: extends stationarySimulationCI
{
    void set_state(const array<real8>/*state*/);
};
```

A nonstationary coupling solved by a Gauss-Seidel- or Jacobi-type scheme will be supported by defining and implementing the interfaces:

```cpp
interface simulationCI: extends stationarySimulationCI
{
    void time_step(const real8/*delta_t*/);
};
```
The interfaces implemented by coFeap (simu.ci) and OpenFOAM (cfdsimu.ci) used in the numerical examples given in this paper are extensions of these interfaces, adding some more special features offered by Feap and oFoam.

4. Conclusions

The question addressed in this paper on software development for efficient solution of multiscale and multiphysics problems is perhaps the most important challenge of the present research. While the paradigm advanced herein on code-coupling to achieve this goal is not necessarily new, since a number of works have focused upon re-using existing codes as a part of a more complex platform, the manner in which it is done at linkage time resulting with truly single code, rather than a wrapper around the executable versions of other codes, is quite original.

We have shown that in order to achieve such a worthy goal, one has to address all the aspects of the problem related not only to software implementation, but also the theoretical formulation and discrete approximation of the multiscale and multiphysics problems. In particular, the crucial role in casting a suitable theoretical formulation is played by the localized Lagrange multipliers. These ideas have been illustrated both for multiscale problems with strong coupling of computations in macroscale and microscale finite elements and multiphysics problems with fluid-structure interaction. The proposed procedure for either case, here presented and discussed for computer code FEAP (Zienkiewicz, Taylor [2005]), can be applied to many other existing codes. In other words, the key ingredient is not as much the kind of code we are using, but rather the ability to formulate the problem with a clear notion of the interface and to split the computations with localized Lagrange multipliers.

The ability to include a number of existing codes within the proposed approach is significantly extended by using special features of the CTL middleware to accept practically all the most often used programming languages in scientific computing, such as C++, Ansi-C and FORTRAN. Moreover, an additional asset is the ability to accept different discrete approximation methods (e.g. finite volume versus finite element), and different time step discretization, and still be able to ensure the stability of computations.

The proposed strategy should accelerate the code development and testing, since it relies on proven codes as the part of the new developments. However, an interesting question remains in particular with respect to multiphysics and multiscale simulations, what will be the limit to a growing number of existing codes incorporated together in this manner. For these future
developments, the hierarchical ordering of simulation codes defined by inheritance of the interfaces they implement, will become interesting. This ordering should define exactly the interchangeability of codes, given by the functionality they offer.

References:

Alexandrescu A. [2001]. Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA


Corba component model, O.M.G. OMG., v4.0 (formal/2006-04-01), 2006


Dubois-Pelerin Y., P. Pegon [1998], Object-oriented programming in nonlinear finite element analysis, Computers and Structures, 67, 225-241


FEAP-homepage [2011], http://www.ce.berkeley.edu/projects/feap/


Hautefeuille M., J.B. Colliat, A. Ibrahimbegovic, H. Matthies, P. Villon [2012], ‘Multiscale approach to modeling inelastic behavior with softening’, Computer and Structures, 94/95, 83-95


Ibrahimbegovic A., B. Brank (eds.) [2005], Multi-physics and multi-scale computer models in nonlinear analysis and optimal design of engineering structures under extreme conditions, IOS Press, Amsterdam

Ibrahimbegovic A., A. Delaplace [2003], Microscale and mesoscale discrete models for dynamic fracture of structures built of brittle materials, Computers and Structures, 81, 1255-1265


Kassiotis C., A. Ibrahimbegovic, R. Niekamp, H. Matthies [2011b], Partitioned solution to nonlinear fluid-structure interaction problems. Part II: CTL based software implementation with nested parallelization, Computational Mechanics, 47, 335-357


Ladeveze P. [2005], Multiscale computational damage modelling of laminate composites, in CISM courses, (ed. T. Sadowski), Springer, 171-212


