An Interface for Code Coupling
ICoCo v1.2 10/05/2010

Fabien Perdu

July 1st, 2010

Rationale

- **Need for flexible coupling algorithms between codes**
  - **Explicit / implicit**
  - **Timesteps of different lengths**
  - **Iterations over one or several timesteps**
  - **Error treatment**
  - **Multigrid (CFD codes)**
  - **Conditional actions (system codes)**
  - **Prediction / correction methods**
  - **Any number of codes**
  - **Large panel of interpolation, data manipulation, …**

- **Multiple loops and conditions depending on**
  - **The code itself**
  - **Another code**
  - **The user / a scenario / physical sensors / …**

Rationale

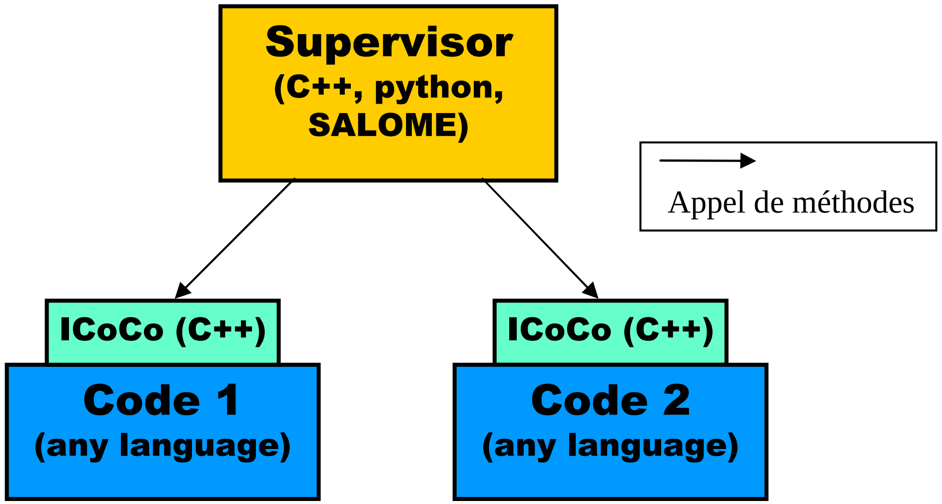
- Two main paradigms
  - Message passing
    - ➢ One code calls performs a send
    - ➢ Another one performs a receive
    - ➢ Both calls MUST match !
    - ⇒ The coupling algorithm (loops, conditions,…) must be written in every code
    - ⇒ Interpolation and data manipulation must be written in one of the codes
  - Method calls (API)
    - ➢ A supervisor performs method calls on every code, following a specified interface (API)
    - ➢ The coupling algorithm is written only once and outside the codes
    - ➢ Interpolation and data manipulation can be performed outside the codes.

Rationale

- **Why chosing the API paradigm ?**
  - **The algorithm is easy to read (located in a single place : the supervisor)**
  - **The algorithm is easy to modify (does not impact the code)**
  - **The algorithm can be reused with other codes.**
- **What is the impact on the codes ?**
  - **They must be modular : the main loop must get out of the code and be flexible.**
  - **Hardly compatible with Fortran-style linear programming.**
- **Why a common API ?**
  - **To make it easy to switch from one code to another one.**
  - **It means less learning to produce a coupled calculation.**
  - **Sharing common rules helps observing them…**

Rationale

- **Overview of the architecture**



General description

- **Scope**
  - **Each code is controllable through a C++ class, deriving from a common mother class named « Problem ».**
  - **Specifications for the codes :**
    ICoCo specifies the methods of the Problem class and what they are supposed to do.
  - **Specifications for the supervisor :**
    ICoCo specifies when it is legal to call each method.
  - **Scope :**
    ICoCo methods allows time advance, saving/restoring and field exchange.

General description

- **Around ICoCo**
  - **Other classes are specified for**
    - **The types of fields exchanged**
    - **The types of exceptions risen.**
- **Use**
  - **The codes implementing the ICoCo interface can be used :**
    - **Directly in C++**
    - **In python via SWIG**
    - **In SALOME via hxx2salome**
- **Parallelism**
  - **For parallel codes, ICoCo methods must be called on all processes with the same arguments, except for fields which are distributed.**

General description

- **Overview of the methods**

  - **Init / End**
    - ➢ **constructor / destructor**
    - ➢ **setDataFile / setMPIComm**
    - ➢ **initialize / terminate**
  - **Time advance**
    - ➢ **presentTime / computeTimeStep**
    - ➢ **initTimeStep / solveTimeStep**
    - ➢ **validateTimeStep / abortTimeStep**
    - ➢ **isStationary**
  - **Sub-iterations**
    - ➢ **iterateTimeStep**

General description

- **Overview of the methods**

  - **Save / Restore**
    - **save / restore / forget**

  - **Field exchange**
    - **getInputFieldsNames / getOutputFieldsNames**
    - **getInputFieldTemplate / setInputField**
    - **getOutputField**

    - **Same methods with MEDCouplingField instead of Field**

Methods specification

- **Init / End**

- `Problem();`

Constructor.

Should not raise any exception.

- `virtual ~Problem();`

Destructor.

Should not raise any exception.

- `virtual void setDataFile(const std::string& datafile);`

Give the name of a datafile to the code.

The call to `setDataFile` **is optional.**

Should be called before `initialize.`

- `virtual void setMPIComm(void* mpicomm);`

Give an MPI communicator to the code, for its internal use.

`mpicomm` **is of type** `void*` **to avoid to include** `mpi.h` **for sequential codes.**

The communicator should include all the processes to be used by the code.

For a sequential run, the call to `setMPIComm` **is optional or** `mpicomm` **should be** `NULL`**.**

Should be called before `initialize.`

Methods specification

- `virtual bool initialize();`

Initialize the code using the arguments of `setDataFile` **and** `setMPIComm`.

This method is called once before any other method.

File reads, memory allocations, and other operations likely to fail should be performed here and not in the previous methods.

It cannot be called again before `terminate`**.**

Return value : true means OK.

If `initialize` **returns false or raises an exception, nothing else than** `terminate` **can be called.**

- `virtual void terminate();`

Terminate the computation, free the memory and save whatever needs to be saved.

This method is called once at the end of the computation or after a non-recoverable error.

After `terminate`**, no method (except** `setDataFile` **and** `setMPIComm`**) can be called before a new call to** `initialize`.

Methods specification

- ## Time advance

The computation time step]t,t+dt] is defined between a sucessful call to `initTimeStep` **and either** `validateTimeStep` **or** `abortTimeStep`.
Only the computation time step can be accessed or modified, but neither the present (t), nor the past.

- `virtual double presentTime() const`
  Returns the current time t.
  Can be called anytime between `initialize` **and** `terminate`**.**
  The current time can only change during the call to `validateTimeStep`.

- `virtual double computeTimeStep(bool& stop) const`
  Returns two data : `stop` **is true if the code wants to stop, and the return value contains the preferred time step for this code.**
  Both data are only indicative, the supervisor is not required to take them into account.
  Can be called whenever the Problem has been initialized but the computation time step is not defined.

Methods specification

- `virtual bool initTimeStep(double dt)`
  Give the next time step to the code.
  Can be called whenever the computation time step is not defined.
  Returns false if `dt` **is not compatible with the code time scheme.**
  After this call (if successful), the computation time step is defined to]t,t+dt] where t is the value which would be returned by `presentTime`**,**
  All input and output fields are allocated on]t,t+dt], initialized, and accessible through field exchange methods.
- `virtual bool solveTimeStep()`
  Perform the computation on the current interval, using input fields.
  Can be called whenever the computation time step is defined.
  Returns false if the computation fails.
  After this call (if successful), the solution on the computation time step is accessible through the output fields.
- `virtual void validateTimeStep()`
  Validate the computation performed by `solveTimeStep`**.**
  Can be called whenever the computation time step is defined.
  After this call, the present time has been advanced to the end of the computation time step, and the computation time step is undefined, so the input and output fields are not accessible any more.

Methods specification

- `virtual void abortTimeStep()`
  Abort the computation on the current timestep.
  Can be called whenever the computation timestep is defined, instead of `validateTimeStep`**.**
  After this call, the present time is left unchanged, and the computation time step is undefined, so the input and output fields are not accessible any more.
- `virtual bool isStationary() const`
  Can be called whenever the computation time step is defined.
  Return value : true if the solution is constant on the computation time step.
  If the solution has not been computed, the return value is of course not meaningful.
- `virtual bool iterateTimeStep(bool& converged)`
  The implementation of this method is optional.
  Perform a single iteration of computation inside the timestep.
  Can be called whenever the computation timestep is defined.
  Returns false if the computation fails. `converged` **is set to true if the solution is not evolving any more.**
  Calling `iterateTimeStep` **until** `converged` **is true is equivalent to calling** `solveTimeStep`**, within the code's convergence threshold.**

Methods specification

- **Save / Restore**

  The save / restore interface is optional.
  It provides the possibility to bring the code back to a previous state.

- `virtual void save(int label, const std::string& method) const`

  Save the state of the code.
  Can be called at any time between `initialize` **and** `terminate`**.**
  The saved state is identified by the couple of `label` **and** `method`**.**
  `method` **is a string specifying which method is used to save the state of the code. A code can provide different methods (for example in memory, on disk,…). At least « default » should be a valid argument.**
  If `save` **has already been called with the same two arguments, the saved state is overwritten.**

- `virtual void restore(int label, const std::string& method)`

  Restore a state previously saved with the same couple of arguments.
  Can be called at any time between `initialize` **and** `terminate`**.**
  After `restore`**, the code should behave exactly like after the corresponding call to** `save`**, except for save/restore methods, since the list of saved states may have changed.**

Methods specification

- `virtual void forget(int label, const std::string& method) const`
  Forget a state previously saved with the same couple of arguments.
  Can be called at any time between `initialize` **and** `terminate`**.**
  After this call, the state cannot be restored anymore.
  It can be used to free the space occupied by unused saved states.

Methods specification

- **Field exchange**

  All field exchange methods can be called whenever the computation timestep is defined.
  The fields must be defined on the computation timestep]t,t+dt].
  Output fields are fields calculated by the code, input fields are provided to the code as, for example, boundary conditions or source terms.

- `virtual std::vector<std::string> getInputFieldsNames() const`

  Return a list of strings identifying input fields.

- `virtual void getInputFieldTemplate(const std::string& name, TrioField& afield) const`

  Get a template of the field expected by the code for a given name.
  This call modifies `afield`**.**
  After filling the values in `afield`**, it can be sent back to the code through the method** `setInputField`**.**
  This method is useful to know the mesh, discretization,… on which an input field is expected.

Methods specification

- `virtual void setInputField(const std::string& name, const TrioField& afield)`
  Provide the input field corresponding to `name` **to the code.**
  After this call, the state of the computation and the output fields are invalidated.
  It should always be possible to switch consecutive calls to `setInputField`**.**
  At least one call to `iterateTimeStep` **or** `solveTimeStep` **must be performed before** `getOutputField` **or** `validateTimeStep` **can be called.**

- `virtual std::vector<std::string> getOutputFieldsNames() const`
  Return a list of strings identifying output fields.

- `virtual void getOutputField(const std::string& name, TrioField& afield) const`
  Get the output field corresponding to `name` **from the code.**
  This call modifies `afield`**.**

Methods specification

Similar methods exist to exchange fields of type ParaMEDMEM::MEDCouplingField instead of ICoCo::TrioField.

- `virtual ParaMEDMEM::MEDCouplingFieldDouble* getInputMEDFieldTemplate(const std::string& name) const;`

- `virtual void setInputMEDField(const std::string& name, const ParaMEDMEM::MEDCouplingFieldDouble* afield);`

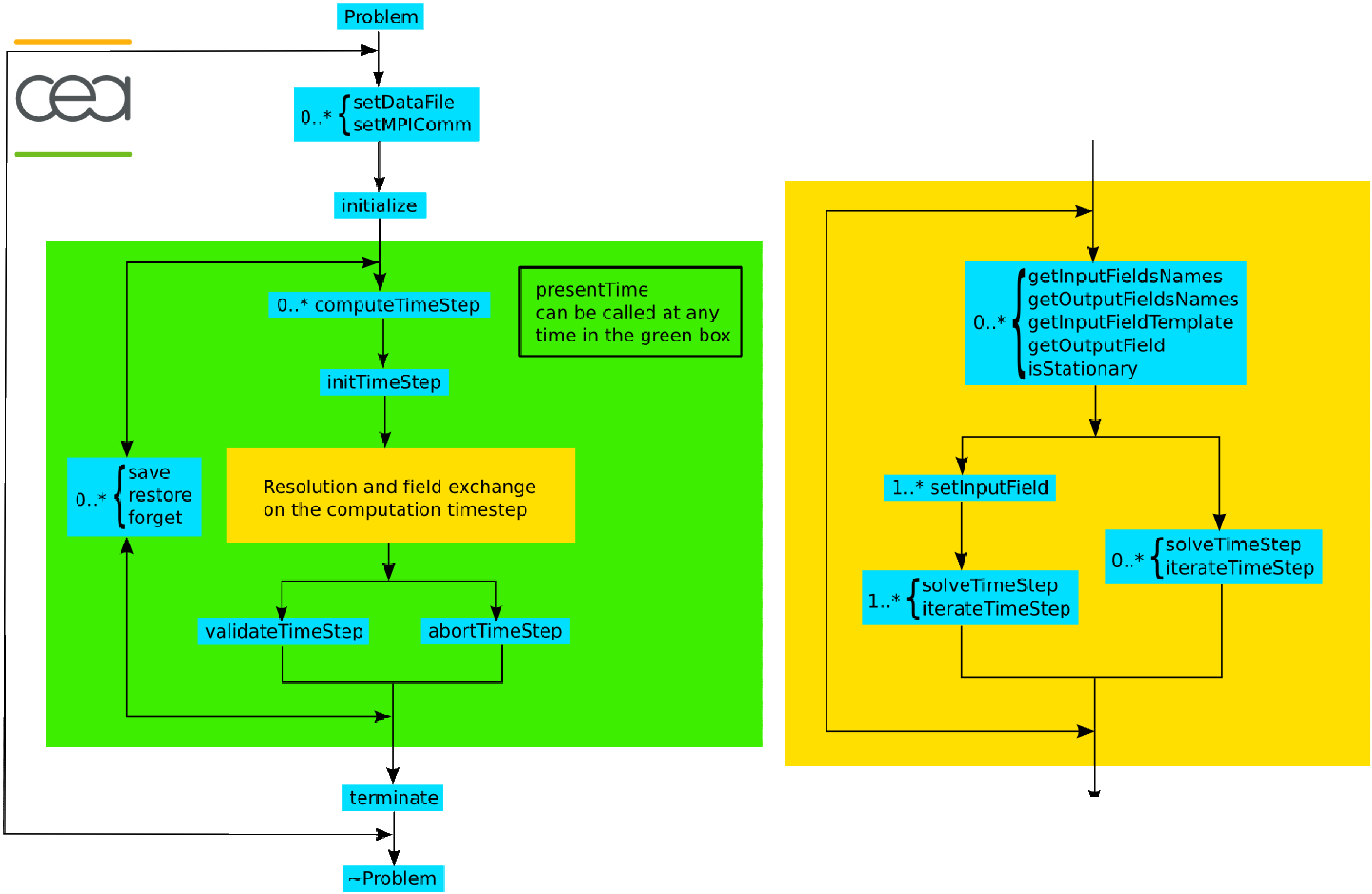
- `virtual ParaMEDMEM::MEDCouplingFieldDouble* getOutputMEDField(const std::string& name) const;`

Finally, the shared library containing the code must provide a way to retrieve the object of type Problem.
This is done through the following function.

- `extern "C" ICoCo::Problem* getProblem()`

  Return a pointer to an object a type Problem implementing ICoCo.
  After use, the object can be freed using the `delete` **operator.**

Execution flow chart



ICoCo evolutions

- **Rate of evolution**
  - **ICoCo interface should be very stable to induce little additional work for the codes.**
  - **Changes only when new features are required.**

- **Backward compatibility**
  - **ICoCo versions should be backward compatible in the following way :**
    - **If a code implements ICoCo version nn**
    - **And a supervisor uses ICoCo version mm**
    - **And nn >= mm**
    - **Then the supervisor should be able to run the code (ideally without recompilation)**
  - **In other words, each version of ICoCo should be a superset of the previous version**

ICoCo evolutions

- **Foreseeable evolutions**
  - **Describe several stages inside a timestep**
    - **With different input fields**
    - **And different output fields**
    - **Depending on one another**
  - **Impose a new mesh to the code**
  - **…**

Thank you for your attention