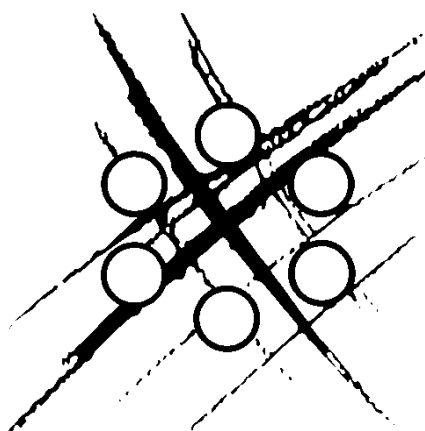




**DIRECTION DE L'ÉNERGIE NUCLEAIRE
DIRECTION DELEGUEE AUX ACTIVITES NUCLEAIRES DE SACLAY
DEPARTEMENT DE MODELISATION DES SYSTEMES ET STRUCTURES
SERVICE DE THERMOHYDRAULIQUE ET MECANIQUE DES FLUIDES**



Documentation of the Interface for Code Coupling : ICoCo

Référence DEN/DANS/DM2S/STMF/LMES/RT/12-029/A

Estelle DEVILLE (LMES), Fabien PERDU(LMSF)

Commissariat à l'énergie atomique et aux énergies alternatives

DEN/DANS/DM2S/STMF/LMES

Centre de Saclay BAT 454 - PC 47

91191 Gif/Yvette cedex - France

Tél. : 01 69 08 66 56 Fax : 01 69 08 52 42 Courriel : marie-claude.rolland@cea.fr

NIVEAU DE CONFIDENTIALITE				
DO	DR	CCEA	CD	SD

PARTENAIRES/CLIENTS	REFERENCE DE L'ACCORD OU DU CONTRAT	TYPE D'ACTION
PROJET NURISP : NUCLEAR REACTOR INTEGRATED SIMULATION PROJECT (COLLABORATIVE PROJECT SEVENTH FRAMEWORK PROGRAM EURATOM)	Contrat numéro 232124	documentation part of D.3.3.1.2 deliverable




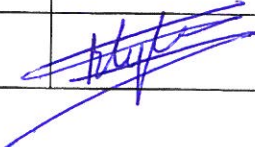
REFERENCES INTERNES CEA			
DIRECTION D'OBJECTIFS	PROGRAMME	PROJET	EOTP

S'IL S'AGIT DU LIVRABLE D'UN JALON :

JALON	INTITULE DU JALON

S'agit-il d'une synthèse ? oui non

SUIVI DES VERSIONS			
INDICE	DATE	NATURE DE L'EVOLUTION	PAGES, CHAPITRES
A		Document initial	

	NOM	FONCTION	VISA	DATE
REDACTEUR	Estelle DEVILLE			05/07/12
	Fabien PERDU			05/07/12
VERIFICATEUR				
APPROBATEUR	Philippe EMONOT	Chef de laboratoire		09/07/12
AUTRE VISA				
ÉMETTEUR	Bernard FAYDIDE	CHEF DE SERVICE		12/07/2012

Documentation of the Interface for Code Coupling : ICoCo

MOTS CLEFS

NURISP, Code Coupling, generic interface

RESUME / CONCLUSIONS

Résumé en français :

Ce rapport reprend le contenu d'un livrable dû dans le cadre du projet européen NURISP, en vue d'une diffusion plus large en interne de la DEN.

Ce livrable D3.3.1.2 documente l'interface générique de couplage de codes développée au CEA et baptisée ICoCo (Interface de Couplage de Codes), et fournit des exemples d'utilisation sur un cas de couplage Cathare-Trio_U soit à partir d'un superviseur C++ (y compris en parallèle) ou python, soit à partir de la plateforme SALOME.

Résumé en anglais :

This document reproduces the content of D.3.3.1.2 deliverable of the NURISP project for a wider diffusion inside DEN. It describes the generic coupling interface developed at CEA and named ICoCo (Interface for Code Coupling), and provides use cases on a Cathare-Trio_U coupling either from a C++ or python supervisor, or from the SALOME platform.

RESUME / CONCLUSIONS de niveau DO en cas de niveau confidentialité supérieur du document

SO

Diffusion initiale interne CEA

Bernard FAYDIDE	DEN/DANS/DM2S/STMf/DIR
Estelle DEVILLE	DEN/DANS/DM2S/STMf/LMES
Fabien PERDU	DEN/DANS/DM2S/STMf/LMSF
Philippe EMONOT	DEN/DANS/DM2S/STMf/LMES
Olivier ANTONI	DEN/DANS/DM2S/STMf/LMSF
Didier JAMET	DEN/DANS/DM2S/STMf/LMEC
Danielle GALO-LEPAGE	DEN/DANS/DM2S/STMf/LATF
Vincent BERGEAUD	DEN/DANS/DM2S/STMf/LGLS
Erwan ADAM	DEN/DANS/DM2S/STMf/LMES
Philippe FILLION	DEN/DANS/DM2S/STMf/LMES
Gilles LAVIALLE	DEN/DANS/DM2S/STMf/LMES
Alain RUBY	DEN/DANS/DM2S/STMf/LMES
Nicolas CROUZET	DEN/DANS/DM2S/STMf/LGLS
Gauthier FAUCHET	DEN/DANS/DM2S/STMf/LGLS
Anthony GEAY	DEN/DANS/DM2S/STMf/LGLS
Nicolas TAVERON	DEN/DANS/DM2S/STMf/LMSF
Simone VANDROUX	DEN/DANS/DM2S/STMf/LMSF
Roland BAVIERE	DEN/DANS/DM2S/STMf/LATF
Jean-Charles LE PALLEC	DEN/DANS/DM2S/SERMA/LPEC
Patrick BLANC-TRANCHANT	DEN/DANS/DM2S/SERMA/DIR
Bruno CHANARON	DEN/DANS/DM2S/DIR
Richard LENAIN	DEN/DANS/DM2S/DIR
Philippe MONTARNAL	DEN/DANS/DM2S/DIR
Jean-Paul DEFFAIN	DEN/EC/DISN/SIMU
Daniel CARUGE	DEN/EC/DISN/SIMU
Jean-Claude GARNIER	DEN/CAD/DER/SESI
David PLANCQ	DEN/CAD/DER/SESI
Frédéric VARAINE	DEN/CAD/DER/SPRC
Emmanuel TOURON	DEN/CAD/DEC/SESC

Diffusion papier :

1 exemplaire DM2S/DIR
 1 exemplaire du résumé pour archivage
 1 exemplaire pour archivage STMf

Diffusion initiale externe CEA

SOMMAIRE

1	Introduction	9
2	Description of ICoCo interface	10
2.1	General description.....	10
2.2	Methods of Problem class	11
2.2.1	Methods for intialization and termination of the code.....	11
2.2.1	Methods for time advance of a code	12
2.2.1	Methods for save and restore state of the code.....	13
2.2.1	Methods for getting fields from a code and setting fields to a code	14
2.3	Types of fields exchanged	15
2.4	Exceptions that could be raised.....	16
2.5	Execution flow chart using ICoCo interface	17
3	Use case: Single phase system/CFD coupling using CATHARE and TRIO-U codes.....	18
4	Use of the generic interface ICoCo with a user developed supervisor.....	22
4.1	ICoCo use with a sequential C++ supervisor	22
4.1.1	Useful functions	22
4.1.2	Instantiation of the problems	25
4.1.3	Initialization of the problems	25
4.1.4	Initialization of coupling parameters and data to transform data.....	25
4.1.5	First time loop : Cathare alone to reach a Cathare steady state.....	26
4.1.6	Second time loop : both codes with one-way coupling	26
4.1.7	End of the coupling calculation.....	30
4.2	ICoCo use with a parallel C++ supervisor	30
4.2.1	Useful functions	31
4.2.2	MPI initialization.....	34
4.2.3	Instantiation of the problems	35
4.2.4	Initialization of the problems	35
4.2.5	Initialization of coupling parameters and data to transform data.....	36
4.2.6	First time loop : Cathare alone to reach a Cathare steady state.....	36
4.2.7	Second time loop : both codes with one-way coupling	37
4.2.8	End of the coupling calculation.....	41
4.3	ICoCo use with a sequential python supervisor	41
4.3.1	Useful functions	41
4.3.2	Swig interface file	44
4.3.3	Instantiation of the problems	45
4.3.4	Initialization of the problems	46
4.3.5	Initialization of coupling parameters and data to transform data.....	46
4.3.6	First time loop : Cathare alone to reach a Cathare steady state.....	47
4.3.7	Second time loop : both codes with one-way coupling	47
4.3.8	End of the coupling calculation.....	51
4.4	ICoCo use with a parallel python supervisor	51

Documentation of the Interface for Code Coupling : ICoCo

4.4.1	Useful functions	51
4.4.2	Swig interface file	56
4.4.3	MPI initialization.....	59
4.4.4	Instantiation of the problems	59
4.4.5	Initialization of the problems	60
4.4.6	Initialization of coupling parameters and data to transform data.....	60
4.4.7	First time loop : Cathare alone to reach a Cathare steady state.....	61
4.4.8	Second time loop : both codes with one-way coupling	61
4.4.9	End of the coupling calculation.....	66
5	Use of the generic interface ICoCo with SALOME platform	67
5.1	Construction of the Salome ICoCo component	67
5.2	Use of the Salome ICoCo component in a python script.....	68
5.2.1	Instantiation of the problems	68
5.2.2	Initialization of the problems	69
5.2.3	Initialization of coupling parameters and data to transform data.....	69
5.2.4	First time loop : Cathare alone to reach a Cathare steady state.....	69
5.2.5	Second time loop : both codes with one-way coupling	70
5.2.6	End of the coupling calculation.....	73
5.3	Use of the Salome ICoCo component in Salome YACS module	74
5.3.1	Instantiation and initialization of the problems.....	75
5.3.2	Initialization of coupling parameters and data to transform data.....	76
5.3.3	First time loop : Cathare alone to reach a Cathare steady state.....	78
5.3.4	Second time loop : both codes with one-way coupling	78
5.3.5	End of the coupling calculation.....	81
Annexes	82
5.4	Annex 1: sources of ICoCo interface.....	82
5.4.1	Header file: Problem.h.....	82
5.4.2	Source file: Problem.cpp	83
5.4.3	Header file: ICoCoField.h	88
5.4.4	Source file: ICoCoField.cpp.....	88
5.4.5	Header field: ICoCoTrioField.h.....	89
5.4.6	Source file: ICoCoTrioField.cpp	90
5.4.7	Header file: Exceptions.h.....	95
5.4.8	Source file: Exceptions.cpp	96
5.5	Annex 2: Main program of a sequential C++ supervisor	97
5.6	Annex 3: Main program of a parallel C++ supervisor	104
5.7	Annex 4: python sequential supervisor.....	114
5.8	Annex 5: python parallel supervisor.....	119
5.9	Annex 6 : header and source file of the Salome ICoCoComponent.....	126
5.9.1	Header file : ICoCoComponent.hxx.....	126
5.9.2	Source File ICoCoComponent.cxx	128
5.10	Annex 7: python script for a Salome use.....	133

liste des figures

Figure 1 : overview of ICoCo architecture	9
Figure 2 : execution flow chart using ICoCo interface.....	17
Figure 3 : scheme of the use case	18
Figure 4 : scheme of the coupling scenario	20
Figure 5 : YACS complete calculation scheme	75
Figure 6 : YACS instantiation and initialization of the problems	76
Figure 7 : YACS initialization of data	77
Figure 8 : YACS first time loop	78
Figure 9 : YACS second time loop.....	79
Figure 10 : YACS end of coupling calculation	81

1 Introduction

This document reproduces the documentation part of D.3.3.1.2 deliverable of the NURISP project which describes the generic coupling interface called ICoCo and provides several use cases.

The ICoCo interface defines how a “Problem” should behave. A Problem is seen like an object which computes a time dependent solution (result of equation solver), function of time dependent input data. The interface specifies methods that the problem has to provide and what they are supposed to do. It also specifies when and how these methods can be called. The supervisor performs the coupling algorithm: it calls methods on every problem and takes in charge interpolation and data manipulation, totally outside the coupled codes.

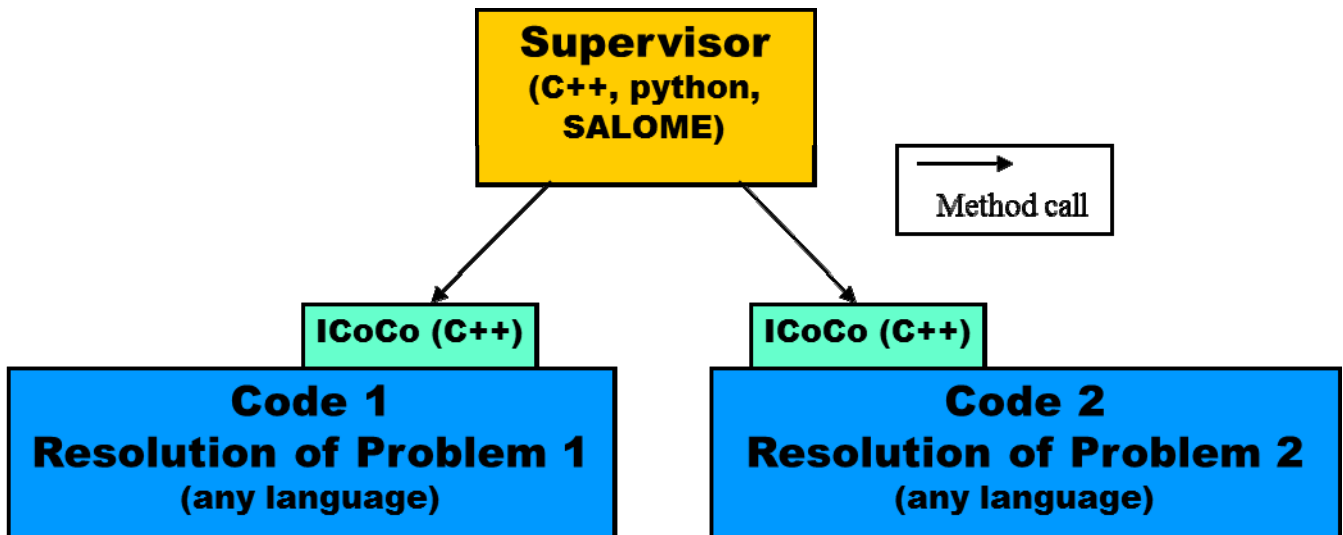


FIGURE 1 : OVERVIEW OF ICOCO ARCHITECTURE

The aim of this document is, first, to describe the generic interface and secondly to detail how to use it through a supervisor which could be provided by SALOME platform or developed by the user (in C++ or in python language).

2 Description of ICoCo interface

2.1 General description

ICoCo is written in C++ and defines mother classes which will control each code.

ICoCo defines methods to a common mother class named "Problem" that allow initialization, time advance, saving and restoring and field exchange. Other classes are also specified for types of fields exchanged and types of exceptions raised.

For parallel codes, ICoCo methods must be called on all processes with the same arguments, except for fields which are distributed.

The shared library containing the code must provide a way to retrieve the object of type Problem.

This is done with the following function `getProblem()` which returns a pointer to an object of type Problem implementing ICoCo. After use, the object can be freed using the *delete* operator.

All header and source files of ICoCo interface are detailed in the second annex.

2.2 Methods of Problem class

The Problem class provides methods for:

- Initialization and termination of the code,
- Time advance and sub-iterations,
- Save and restore,
- Field exchange.

The following paragraphs will detail the signature of all methods of Problem class and also precise which method is optional and in which order they have to be called.

2.2.1 **Methods for initialization and termination of the code**

The table below sums up necessary information on initialization and termination methods.

Signature of the method	Description
<i>Problem()</i>	Default constructor
<i>virtual ~Problem()</i>	Destructor
<i>virtual void setDataFile(const std::string& datafile)</i>	Give the name of a data file to the code The call is optional If called, has to be done before <i>initialize</i>
<i>virtual void setMPIComm(void* mpicomm)</i>	Give an MPI communicator to the code, for its internal use. <i>mpicomm</i> is of type <i>void*</i> to avoid to include <i>mpi.h</i> for sequential codes. The communicator should include all the processes to be used by the code. For a sequential run, the call to <i>setMPIComm</i> is optional or <i>mpicomm</i> should be <i>NULL</i> . Should be called before <i>initialize</i> .
<i>virtual bool initialize()</i>	Initialize the code using the arguments of <i>setDataFile</i> and <i>setMPIComm</i> . This method is called once before any other method. File reads, memory allocations, and other operations which are likely to fail should be performed here and not in the previous methods. It cannot be called again before <i>terminate</i> has been performed. Return value: true means OK. If <i>initialize</i> returns <i>false</i> or raises an exception, nothing else than <i>terminate</i> can be called.
<i>virtual bool terminate()</i>	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 <i>terminate</i> , no method (except <i>setDataFile</i> and <i>setMPIComm</i>) can be called before a new call to <i>initialize</i> .

Documentation of the Interface for Code Coupling : ICoCo
2.2.1 Methods for time advance of a code

The computation time step $[t, t+dt]$ is defined between a successful 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.

The table below sums up necessary information on time advance methods.

Signature of the method	Description
<i>virtual double presentTime() const</i>	Returns the current time t . Can be called anytime between <i>initialize</i> and <i>terminate</i> . The current time can only change during the call to <i>validateTimeStep</i> .
<i>virtual double computeTimeStep(bool& stop) const</i>	Returns two data : <i>stop</i> 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 <i>Problem</i> has been initialized but the computation time step is not defined.
<i>virtual bool initTimeStep(double dt)</i>	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 <i>presentTime</i> . All input and output fields are allocated on $[t, t+dt]$, initialized, and accessible through field exchange methods.
<i>virtual bool solveTimeStep()</i>	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.
<i>virtual void validateTimeStep()</i>	Validate the computation performed by <i>solveTimeStep</i> . 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.

Documentation of the Interface for Code Coupling : ICoCo

<i>virtual void abortTimeStep()</i>	Abort the computation on the current time-step. Can be called whenever the computation time-step is defined, instead of <i>validateTimeStep</i> . 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.
<i>virtual bool isStationary() const</i>	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.
<i>virtual bool iterateTimeStep(bool& converged)</i>	The implementation of this method is optional. Perform a single iteration of computation inside the time-step. Can be called whenever the computation time-step is defined. Returns false if the computation fails. <i>converged</i> is set to true if the solution is not evolving any more. Calling <i>iterateTimeStep</i> until <i>converged</i> is true is equivalent to calling <i>solveTimeStep</i> , within the code's convergence threshold.

2.2.1 Methods for save and restore state of the code

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

The table below sums up necessary information on save and restore methods.

Signature of the method	Description
<i>virtual void save(int label, const std::string& method) const</i>	Save the state of the code. Can be called at any time between <i>initialize</i> and <i>terminate</i> . The saved state is identified by the couple of <i>label</i> and <i>method</i> . <i>method</i> 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 <i>save</i> has already been called with the same two arguments, the saved state is overwritten.
<i>virtual void restore(int label, const std::string& method)</i>	Restore a state previously saved with the same couple of arguments. Can be called at any time between <i>initialize</i> and <i>terminate</i> . After <i>restore</i> , the code should behave exactly like after the corresponding call to <i>save</i> , except

Documentation of the Interface for Code Coupling : ICoCo

virtual void forget(int label, const std::string& method) const

for save/restore methods, since the list of saved states may have changed.

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.

2.2.1 Methods for getting fields from a code and setting fields to a code

All field exchange methods can be called whenever the computation time-step is defined.

The fields must be defined on the computation time-step $[t, t+dt]$.

Output fields are fields calculated by the code, input fields are provided to the code as, for example, boundary conditions or source terms.

Similar methods exist to exchange either fields of type `ICoCo::TrioField` (delivered with `ICoCo`, will be seen in the following paragraph), or fields of type `ParaMEDMEM::MEDCouplingField` delivered by SALOME platform.

`ICoCo` decided to develop the `ICoCo::TrioField` class to be more efficient at coupling the CFD-code TRIO-U and system code CATHARE.

The table below sums up necessary information on fields exchange.

Signature of the method	Description
<i>virtual std::vector<std::string> getInputFieldsNames() const</i>	Return a list of strings identifying input fields.
<i>virtual void getInputFieldTemplate(const std::string& name, TrioField& afield) const</i>	Get a template of the field expected by the code for a given <i>name</i> . This call modifies <i>afield</i> . After filling the values in <i>afield</i> , it can be sent back to the code through the method <i>setInputField</i> . This method is useful to know the mesh, discretization... on which an input field is expected. The <i>TrioField</i> class will be described in a following paragraph.
<i>virtual ParaMEDMEM::MEDCouplingFieldDouble* getInputMEDFieldTemplate(const std::string& name) const</i>	Get a template of the field expected by the code for a given <i>name</i> . Return a <code>ParaMEDMEM::MEDCouplingFieldDouble</code> field
<i>virtual void setInputField(const std::string& name, const TrioField& afield)</i>	Provide the input field corresponding to <i>name</i> 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 <i>setInputField</i> . At least one call to <i>iterateTimeStep</i> or

Documentation of the Interface for Code Coupling : ICoCo

<pre>virtual void setInputMEDField(const std::string& name, const ParaMEDMEM::MEDCouplingFieldDouble* afield)</pre>	<p><i>solveTimeStep</i> must be performed before <i>getOutputField</i> or <i>validateTimeStep</i> can be called.</p> <p>Provide the input field corresponding to <i>name</i> 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 <i>setInputField</i>. At least one call to <i>iterateTimeStep</i> or <i>solveTimeStep</i> must be performed before <i>getOutputField</i> or <i>validateTimeStep</i> can be called.</p>
<pre>virtual std::vector<std::string> getOutputFieldsNames() const virtual void getOutputField(const std::string& name, TrioField& afield) const</pre>	<p>Return a list of strings identifying output fields.</p> <p>Gets the output field corresponding to <i>name</i> from the code. This call modifies <i>afield</i>.</p>
<pre>virtual ParaMEDMEM::MEDCouplingFieldDouble* getOutputMEDField(const std::string& name) const</pre>	<p>Return the output field corresponding to <i>name</i> from the code.</p>

2.3 Types of fields exchanged

The supervisor can exchange either fields of type ICoCo::TrioField or fields of type ParaMEDMEM::MEDCouplingField delivered by SALOME platform.

ICoCo decided to develop the *TrioField* class to be more efficient at coupling the CFD-code TRIO-U and system code CATHARE.

The *TrioField* structure contains all the necessary information to construct a ParaMEDMEM::ParaField field (with the addition of the MPI communicator). This structure can either own or not the values.

The *TrioField* class is delivered with ICoCo interface. The table below gives signatures and descriptions of methods of *TrioField* class.

Signature of the method	Description
<i>TrioField</i> ()	Constructor
<i>TrioField</i> (const <i>TrioField</i> & <i>OtherField</i>)	Copy constructor
~ <i>TrioField</i> ()	Destructor
void <i>clear</i> ()	After the call to <i>clear</i> , all pointers are null and field ownership is false. Arrays are deleted if necessary
void <i>set_standalone</i> ()	After the call to <i>set_standalone</i> , field ownership is true and field is allocated to the right size. The values of the field have been copied if necessary.
void <i>dummy_geom</i> ()	Used to simulate a 0D geometry (Cathare/Trio for example)
<i>TrioField</i> & <i>operator</i> =(const <i>TrioField</i> &	Overload operator = for <i>TrioField</i> . This

Documentation of the Interface for Code Coupling : ICoCo

<i>NewField</i>)	becomes an exact copy of <i>NewField</i> . If <i>NewField</i> ownership is false, they point to the same values. Otherwise the values are copied.
<i>void save(std::ostream& os) const</i>	Save the field to a .field file
<i>void restore(std::istream& in)</i>	Restore field from a .field file
<i>int nb_values() const</i>	Return the number of value locations

2.4 Exceptions that could be raised

ICoCo interface deliver the following exceptions:

- *WrongContext* : raised when a method is called and it is not permitted (another one should be called before for example)
- *WrongArgument* : raised when a method is called with a wrong argument (with a non-recognized value)
- *NotImplemented* : raised when a code has not implemented one of the defined methods in the interface

2.5 Execution flow chart using ICoCo interface

The chart below shows the sequence of method's call. Remind that after the *setInputField* method, at least one call to *iterateTimeStep* or *solveTimeStep* must be performed before *getOutputField* or *validateTimeStep* can be called.

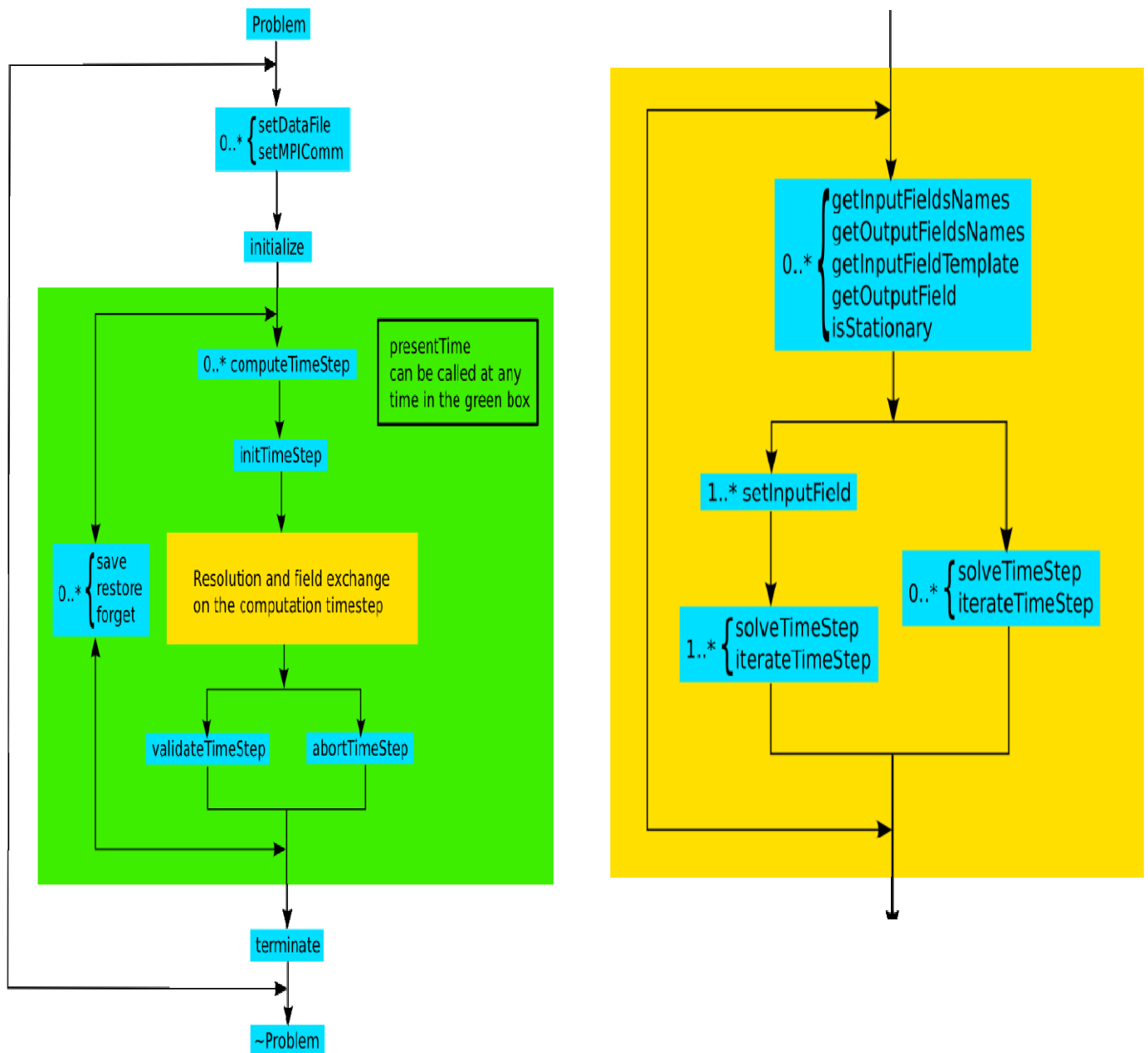


FIGURE 2 : EXECUTION FLOW CHART USING ICOCO INTERFACE

3 Use case: Single phase system/CFD coupling using CATHARE and TRIO-U codes

The use case is a simple one, just to illustrate how to exchange variables between two codes. It represents a two loops circuit connected to a core.

The scheme of the circuit is given below.

Each loop contains:

- A momentum source term in order to model the primary pump
- Pressure drop along the pipe to calculate wall friction
- A heat exchange at the wall in order to represent the heat exchanger

Each loop is connected to the same thermal source term in order to represent the core.

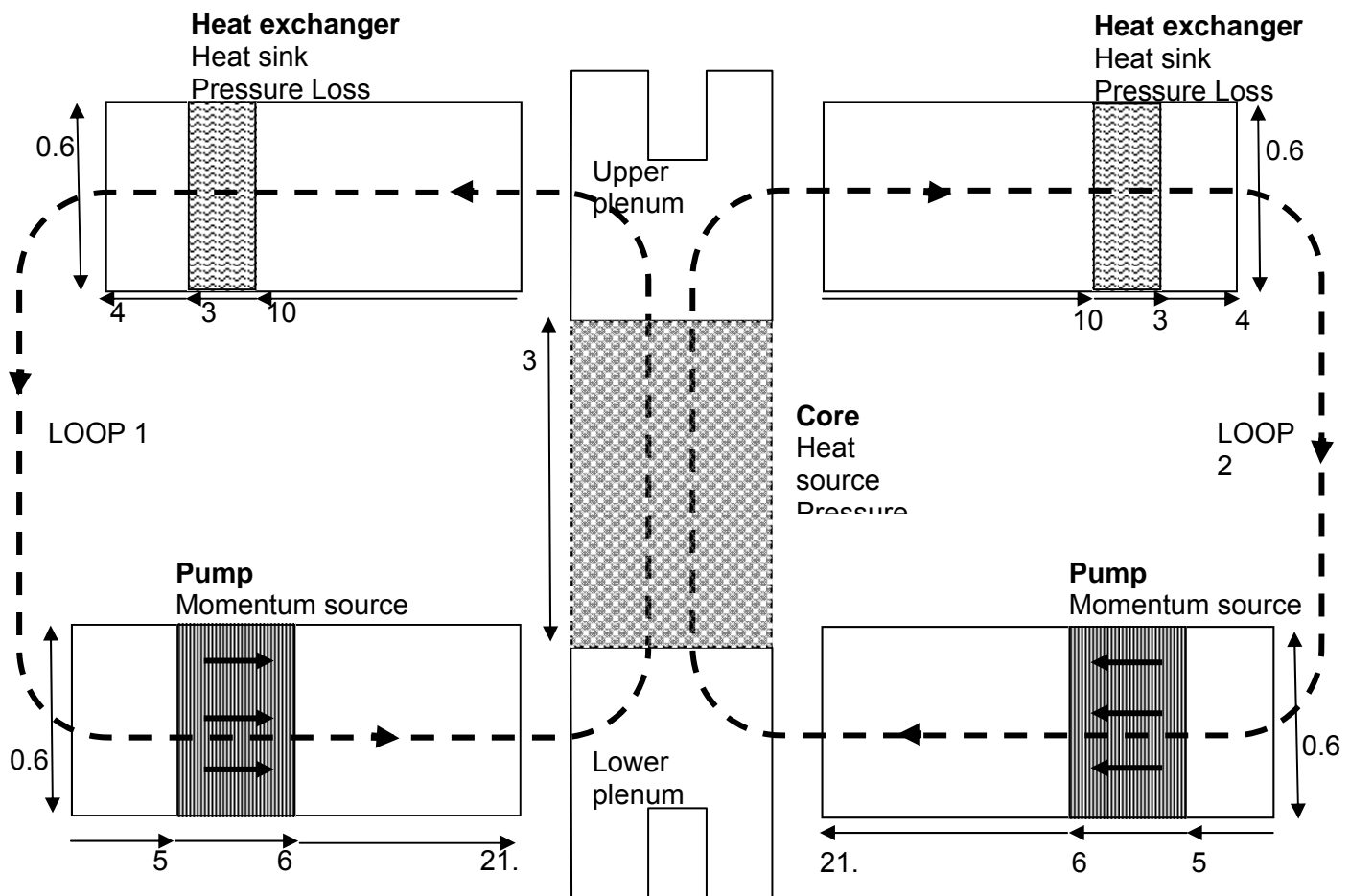


FIGURE 3 : SCHEME OF THE USE CASE

Each loop is supposed to have a constant diameter ($D_{\text{pipe}}=0.6\text{m}$) and a constant section $A_{\text{pipe}}=\pi \cdot (D_{\text{pipe}}/2)^2$. The first loop contains a 'pressurizer' modeled by a fixed pressure ($150 \cdot 10^5 \text{ Pa}$).

Primary pump:

The primary pump is modeled by a momentum source term giving a pressure difference between the inlet and the outlet of the 'pump' of $\Delta P = 1.17 \cdot 10^5$ Pa.

The length of this part of the loop is $L_p = 1$ m

The hydraulic diameter is $D_h = D_{\text{pipe}}$.

The value of the source term is obtained by the following relation: $\Delta P \cdot A_{\text{pipe}} / L_p$

Core

The core is modeled by a heat source term equal to 300 MW

The length of this part of the loop is $L_c = 3$ m.

In order to calculate a proper pressure decrease in the core, the hydraulic diameter (used only for the wall friction) is modified. It is equal to $4 \times 2.3 \cdot 10^{-3}$ m. This value is based on the core geometry.

Heat exchanger

The heat exchanger is modeled by an exchange through a wall with a given outside temperature.

The length of this part of the loop is $L_{\text{hex}} = 3$ m.

The outside wall temperature is 200°C. The wall heat exchange coefficient is chosen to be equal to $h_{\text{wall}} = 5000$ W.m².C⁻¹.

In order to calculate a proper pressure decrease in the heat exchanger, the hydraulic diameter (used only for the wall friction and heat exchange surface) is modified. It is equal to 0.01 m. This value is based on the heat exchanger geometry.

In the coupling scenario, the circuit is modeled by CATHARE except the core which is modeled by Trio_U as shown in the figure below.

Documentation of the Interface for Code Coupling : ICoCo

In the previous figure, the blue part shows what is calculated by CATHARE and the red part shows what is calculated by Trio_U. The exchange is made at the boundaries between the two codes. In this figure, the black arrows show the flow direction. The blue arrows show the data taken from Cathare and given to Trio_U, and the red arrows show the data taken from Trio_U and given to Cathare.

This paragraph will not detail the coupling methodology; it's not the scope of this report so one just gives information on how data fields are exchanged.

Following data are calculated by Cathare and given to Trio_U:

- The mass flow at each boundary between the two codes comes from Cathare
- Cathare gives the enthalpies at the boundaries; they are converted into temperatures which are given to Trio_U. They are used only at the boundaries where the gas flows towards the flow domain

These are data given by Trio_U to Cathare: the feedbacks:

- Trio_U gives the temperatures at the boundaries; they are converted into enthalpies which are given to Cathare. They are used only at the boundaries where the gas flow outwards the Trio_U domain.
- The pressure at one of the Trio_U boundaries is taken as the reference. At the other boundaries, the pressure drop between the reference pressure and the pressure at the boundary is calculated and imposed to Cathare

The algorithm is briefly described here and runs as follows:

- The Cathare and Trio_U problems are initialized
- The different coupling parameters are initialized
 - Time during which Cathare runs alone
 - Time at which the Trio_U to Cathare Pressure feedback begins
 - Time at which the Trio_U to Cathare temperature feedback begins
 - Data to convert enthalpies into temperatures and vice-versa
- Cathare runs alone until it reaches a coherent steady state
- Cathare and Trio_U run together with the boundary conditions (mass flow and enthalpy) taken from Cathare and transformed into data usable by Trio_U (enthalpy transformed into a temperature) but without feedback from Trio_U until Trio_U reaches a steady state.
- Once Trio_U has reached a steady state, the pressure and temperature boundary conditions of the TRIO_U calculation domain are obtained. These data are used to calculate Cathare input data (momentum source and enthalpy).
- Cathare and Trio_U run together with feedback from Trio_U to Cathare. The feedbacks are set progressively to avoid pressure shocks.
- At the end of the calculation, both codes are stopped, the objects are freed.

The time advance is managed as follows. The time-step is chosen as the minimum value between Cathare time-step and Trio_U time-step. The coupling is explicit: the physical data are exchanged once before each time-step and the time-step calculation is performed only once, except if one code fails, in which case both codes abort it and try with a smaller one. If one code wants to stop the calculation (maximum time reached, for example), both codes are stopped.

This algorithm will be used with user developed supervisors (§4.1, §4.2 and §4.3) and with Salome platform (§4.4).

4 Use of the generic interface ICoCo with a user developed supervisor

The user can develop its supervisor either in C++ language or in Python language. This last case is possible with SWIG tool.

THE SUPERVISOR FOLLOWS THE EXECUTION FLOW CHART (

Figure 2) and the algorithm is the same in all the cases (§3).

First of all, the supervisor has to instantiate the ICoCo problems which will control execution of the codes that have to be coupled.

Then the supervisor can initialize the different problems with:

- *setDataFile*, if needed
- *setMPIComm*, if needed
- *initialize*

The supervisor starts the time loop which will end after reaching the final time.

The computation time step is performed by *computeTimeStep* method, called for each interfaced code. The supervisor can use different time step for each code or use the same one (the minimum one for example). It depends on the coupling strategy.

The elected computation time step is given to the code with *initTimeStep* method (same one for all codes or a different one by code).

The supervisor recovers the output fields to be exchanged from one code to another with *getOutputField* method. If needed, the fields can be changed before being injected into wanted code.

The input fields can now be passed to the codes through *setInputField* methods (after a call of *getInputFieldTemplate*).

The computations of each code are now performed on the current interval, using modified input fields through *solveTimeStep*.

If everything is OK for each code, the computation can be validated through *validateTimeStep* method.

If there is a problem with one of the code, the computations are stopped through *abortTimeStep* method and another computation time step has to be performed and given to all codes.

Once the final time reached, the supervisor closes all the problems with:

- *terminate*
- *destructor*

4.1 ICoCo use with a sequential C++ supervisor

To make the comprehension of the writing of a C++ supervisor easier, the writing of the main program will be cut out in small steps. The whole main program can be found in the third annex.

4.1.1 Useful functions

Due to the fact that we couple two codes and exchange fields between these two codes, some functions can be written to avoid code duplication.

These functions concern:

Documentation of the Interface for Code Coupling : ICoCo

- To instantiate the problems, if the C++ supervisor is linked to the codes, it has just to call to the *new Problem* methods. If it's not, the supervisor accesses the problems through the *getProblem* function (opens the code dynamic library with *dlopen* method and gets the function with *dlsym* method). This is done in *openLib* function. At the end, remind that the code dynamic library should be closed. This is the aim of the *closeLib* function.
- To get a one component field from a problem and to return the mean value, the *getFieldMean* function is written. This is necessary because a Cathare cell is connected to a *Trio_U* field on multiple cells
- To set a one component field to a problem with the same value affected to the field cells, the function *setConstantField* is written.

```

#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>

using namespace std;
using namespace ICoCo;

// OpenLib function
// Input parameter: the name of the library
// output parameter: a pointer to the problem
// Inout parameter : an opaque "handle" for the dynamic library
Problem* openLib(const char* libname, void* &handle) {
    // open shared library
    Problem *(*getProblem)();
    // open the code dynamic library
    handle =dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
        cerr << dlerror() << endl;
        throw 0;
    }
    // look at getProblem method in the code dynamic library
    getProblem=(Problem* (*)())dlsym(handle, "getProblem");
    if (!getProblem) {
        cout << dlerror() << endl;
        throw 0;
    }
    // instantiation of ICoCo Problem
    return (*getProblem)();
}

// closeLib function
// Input parameter: an opaque "handle" for the dynamic library
void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

```

Documentation of the Interface for Code Coupling : ICoCo

```
// getFieldMean function
// input parameters: 1- pointer to ICoCo Problem which provides the output
// field
//                               2- name of the field to get
// output parameter: the mean of the field (a double)
double getFieldMean(Problem *P, string name) {
    double mean=0.;
    TrioField f ;

    // get the output field from the problem
    P->getOutputField(name,f);
    // only one component fields are treated
    if (f._nb_field_components!=1)
        throw 0;

    // Compute and return the mean value (all elements/nodes have the same
    // weight)
    for (int loc=0;loc<f.nb_values();loc++)
        mean=mean+f._field[loc];
    return mean/f.nb_values();
}

// setConstantField function
// input parameters: 1- pointer to ICoCo Problem to which the field will be
// set
//                               2- name of the field to set
//                               3- value to set
//                               4- component of the field to affect (1st one by default)
// output parameter: No output parameter

void setConstantField(Problem *P, string name, double val, int comp=0) {
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();

    // Fill the values
    // Give a constant field to a Problem.
    // One component of the field is specified, the others are 0.
    int nb_elems=f._nb_elems;
    int nb_comp=f._nb_field_components;
    for (int i=0;i<nb_elems;i++)
        for (int j=0;j<nb_comp;j++)
            if (j!=comp)
                f._field[i*nb_comp+j]=0;
            else
                f._field[i*nb_comp+j]=val;

    // Give the field to the problem
    P->setInputField(name,f);
}
```


4.1.2 Instantiation of the problems

For each code, a call to openLib function described in previous paragraph is enough.

```
int main(int argc, char** argv) {  
  
    // open TRIO-U shared library and instantiate ICoCo Trio_U Problem T  
    void* handle_trio;  
    Problem *T=openLib("_Trio_UModule_opt.so", handle_trio);  
  
    // open CATHARE shared library and instantiate ICoCo Cathare Problem C  
    void* handle_cathare;  
    Problem *C=openLib("./libcathare_gad.so", handle_cathare);  
}
```

4.1.3 Initialization of the problems

Trio_U needs both data file and a MPI communicator (even in sequential).

For Cathare code, it is quite different.

In fact, before doing a Cathare calculation process, Cathare needs a pre-processing step where data acquisition is done (description of the hydraulic circuit(s) to be simulated, the events occurring during the simulation and how calculation is managed). This pre-processing step is done and the pre-compiled data file is integrated to the dynamic library loaded at problem instantiation.

Then, the data file has already been defined through the dynamic library loaded at problem instantiation. As a consequence, Cathare doesn't need any initialization parameter.

```
// initialization of the problems  
T->setDataFile('jdd_Trio.data');  
T->setMPIComm(0) ; //sequential calculation  
T->initialize() ;  
C->initialize();
```

4.1.4 Initialization of coupling parameters and data to transform data

```
// Coupling parameters  
// Time for which Cathare runs alone. Allows Cathare to reach a steady  
state.  
double t_begin_trio=1000;  
// Time at which Trio_U -> Cathare retroaction on pressure begins  
double time_retro_P=1001;  
// Time at which Trio_U -> Cathare retroaction on temperature begins  
double time_retro_T=1003;  
  
// data to transform Cathare enthalpy into Trio_U temperature or vice versa  
//  $H = href + Cp\_Trio * ( T - Tref )$   
double href = 1.15116e6 ;  
double tref = 263.8+273.15 ; // Trio_U temperature are in K  
double Cp_Trio = 5332.43 ;  
  
// mass flowrate transformation
```

Documentation of the Interface for Code Coupling : ICoCo

```
// In Cathare, pipe of diameter 0.6m (S=0.28274m²)
// In Trio_U, 2D pipe of section 0.6m
// We keep the same Q=rho*u*S between Cathare and Trio_U
// => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
// !! u(Trio) != u(Cathare)
double S_Trio = 0.6 ;
double rho_Trio = 739.206 ;

// variables for pressure retroaction
// Must be remembered from timestep to timestep
double dP1=0; // outlet1 - inlet1
double dP2=0; // outlet2 - inlet1
double dP3=0; // inlet2 - inlet1
```

4.1.5 First time loop : Cathare alone to reach a Cathare steady state

During the first 1000 seconds, Cathare will run alone to reach a Cathare steady state.

```
// First time loop (Cathare alone)
double epsilon=1e-10; // small time for double comparisons
bool stop; // Does the problem want to stop ?

while (1) { // Loop on timesteps
    double present=C->presentTime();
    double dt=C->computeTimeStep(stop);
    if (stop || present>t_begin_trio-epsilon)
        break;

    // Modify dt in order to come to exactly t_begin_trio
    if (present+dt>t_begin_trio)
        dt=t_begin_trio-present;
    else if (present+2*dt>t_begin_trio)
        dt=(t_begin_trio-present)/2;

    // Initialize the timestep
    C->initTimeStep(dt);

    // Perform the computation
    bool ok=C->solveTimeStep();

    // Either validate or abort it
    if (!ok) // The resolution failed, try with a new dt
        C->abortTimeStep();
    else // Validate and go to the next time step
        C->validateTimeStep();
} // End loop on timesteps
```

4.1.6 Second time loop : both codes with one-way coupling

This part contains computation of the coupling time step, initialization of the time step for the two codes, getting output from Cathare, and Trio_U, giving input to Trio_U and Cathare, launching the codes and go away if everything OK until final time step has been reached.

Documentation of the Interface for Code Coupling : ICoCo

Following data are calculated by Cathare and given to Trio_U:

- The mass flow at each boundary between the two codes comes from Cathare
- Cathare gives the enthalpies at the boundaries, they are converted into temperatures which are given to Trio_U. They are used only at the boundaries where the gas flows towards the flow domain

These are data given by Trio_U to Cathare: the feedbacks :

- Trio_U gives the temperatures at the boundaries, they are converted into enthalpies which are given to Cathare. They are used only at the boundaries where the gas flow outwards the Trio_U domain.
- The pressure at one of the Trio_U boundaries is taken as the reference. At the other boundaries, the pressure drop between the reference pressure and the pressure at the boundary is calculated and imposed to Cathare

```
// Second time loop (both codes)
bool ok=true;      // Is the time interval successfully solved?
stop=false;

while (!stop) {           // Loop on timesteps
    ok=false;
    while (!ok) {         // Loop on timestep size
        bool stop_T,stop_C;

        // Compute time step length
        double dt_C=C->computeTimeStep(stop_C);
        double dt_T=T->computeTimeStep(stop_T);
        double dt=min(dt_C,dt_T);

        // And decide if we have to stop
        stop = stop_T || stop_C;
        if (stop)
            break;

        //prepare the new time step
        C->initTimeStep(dt);
        T->initTimeStep(dt);

        // Get output from Cathare
        double C_pressure_inlet1, C_pressure_outlet1;
        double C_pressure_inlet2, C_pressure_outlet2;
        double C_massflow_inlet1, C_massflow_outlet1;
        double C_massflow_inlet2, C_massflow_outlet2;
        double C_enthalpy_inlet1, C_enthalpy_outlet1;
        double C_enthalpy_inlet2, C_enthalpy_outlet2;

        // Pressures
        C_pressure_inlet1=getFieldMean(C,"PRESSURE_PIPE11_85");
        C_pressure_outlet1=getFieldMean(C,"PRESSURE_PIPE12_1");
        C_pressure_inlet2=getFieldMean(C,"PRESSURE_PIPE21_85");
        C_pressure_outlet2=getFieldMean(C,"PRESSURE_PIPE22_1");
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Mass flow rates
C_massflow_inlet1=getFieldMean(C,"LIQFLOW_PIPE11_85");
C_massflow_outlet1=getFieldMean(C,"LIQFLOW_PIPE12_1");
C_massflow_inlet2=getFieldMean(C,"LIQFLOW_PIPE21_85");
C_massflow_outlet2=getFieldMean(C,"LIQFLOW_PIPE22_1");

// Enthalpies
C_enthalpy_inlet1=getFieldMean(C,"LIQH_PIPE11_85");
C_enthalpy_outlet1=getFieldMean(C,"LIQH_PIPE12_1");
C_enthalpy_inlet2=getFieldMean(C,"LIQH_PIPE21_85");
C_enthalpy_outlet2=getFieldMean(C,"LIQH_PIPE22_1");

// Compute Trio_U input values
//calculation of input temperature for Trio_U
double temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref;
double temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref;
double temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref;
double temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref;

// calculation of inlet and outlet velocity for Trio_U
double u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio;
double u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio;
double u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio;
double u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio;

// Modifications to ensure div(u)=0
// works because all areas are equal.
double divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2;
u_outlet1-=divergence/4;
u_outlet2-=divergence/4;
u_inlet1+=divergence/4;
u_inlet2+=divergence/4;

// Gets output from Trio_U
double T_pressure_inlet1, T_pressure_inlet2 ;
double T_pressure_outlet1, T_pressure_outlet2;
double T_temperature_inlet1, T_temperature_inlet2 ;
double T_temperature_outlet1, T_temperature_outlet2;

// Pressures
T_pressure_inlet1=getFieldMean(T,"pressure_inlet1")*rho_Trio;
T_pressure_inlet2=getFieldMean(T,"pressure_inlet2")*rho_Trio;
T_pressure_outlet1=getFieldMean(T,"pressure_outlet1")*rho_Trio;
T_pressure_outlet2=getFieldMean(T,"pressure_outlet2")*rho_Trio;

// Temperatures
T_temperature_inlet1=getFieldMean(T,"temperature_inlet1");
T_temperature_inlet2=getFieldMean(T,"temperature_inlet2");
T_temperature_outlet1=getFieldMean(T,"temperature_outlet1");
T_temperature_outlet2=getFieldMean(T,"temperature_outlet2");

// Give input to Cathare
```

Documentation of the Interface for Code Coupling : ICoCo

```

// Trio_U -> Cathare retroaction on pressure
if (C->presentTime()>time_retro_P) {
    double dP_T1 = T_pressure_inlet1-T_pressure_outlet1;
    double dP_T2 = T_pressure_inlet1-T_pressure_outlet2;
    double dP_T3 = T_pressure_inlet1-T_pressure_inlet2;

    double dP_C1 = C_pressure_inlet1-C_pressure_outlet1;
    double dP_C2 = C_pressure_inlet1-C_pressure_outlet2;
    double dP_C3 = C_pressure_inlet1-C_pressure_inlet2;

    double tau_coupl=0.1;
    double k_loc=0;
    if (dt>tau_coupl)
        k_loc=1;
    else
        k_loc=dt/tau_coupl;

    // pressure retroaction on outlets
    dP1 += (dP_C1-dP_T1)*k_loc;
    dP2 += (dP_C2-dP_T2)*k_loc;
    dP3 -= (dP_C3-dP_T3)*k_loc;

    setConstantField(C,"DPLEXT_PIPE12_1",dP1);
    setConstantField(C,"DPLEXT_PIPE22_1",dP2);
    setConstantField(C,"DPLEXT_PIPE21_86",dP3);
}

// Trio_U -> Cathare retroaction on temperature
if (C->presentTime()>time_retro_T) {
    double T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref);
    double T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref);
    double T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref);
    double T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref);

    // mesh numbers are growing from "left" to "right"
    // overlapping domain on the "right"
    // Loop 1
    setConstantField(C,"OVHLEXT_PIPE11_85",-10);
    setConstantField(C,"OVPLEXT_PIPE11_85",-10);
    setConstantField(C,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1);
    // Loop 2
    setConstantField(C,"OVHLEXT_PIPE21_85",-10);
    setConstantField(C,"OVPLEXT_PIPE21_85",-10);
    setConstantField(C,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2);

    // overlapping domain on the "left"
    // Loop 1
    setConstantField(C,"OVHLEXT_PIPE12_1",10);
    setConstantField(C,"OVPLEXT_PIPE12_2",10);
    setConstantField(C,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1);
    // Loop 2
    setConstantField(C,"OVHLEXT_PIPE22_1",10);
    setConstantField(C,"OVPLEXT_PIPE22_2",10);

```

Documentation of the Interface for Code Coupling : ICoCo

```

        setConstantField(C,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2);
    }

    // Give input to Trio_U
    // Velocities
    setConstantField(T,"u_inlet1",u_inlet1,1);
    setConstantField(T,"u_inlet2",u_inlet2,1);
    setConstantField(T,"u_outlet1",u_outlet1,1);
    setConstantField(T,"u_outlet2",u_outlet2,1);

    // Temperatures
    setConstantField(T,"temperature_inlet1",temperature_inlet1);
    setConstantField(T,"temperature_inlet2",temperature_inlet2);
    setConstantField(T,"temperature_outlet1",temperature_outlet1);
    setConstantField(T,"temperature_outlet2",temperature_outlet2);

    // Solve next time step
    bool ok_T=T->solveTimeStep();
    bool ok_C=C->solveTimeStep();
    ok = ok_T && ok_C;

    // Two ways : The resolution failed, try with a new dt or validate and
go // to next time step
    if (!ok) {
        T->abortTimeStep();
        C->abortTimeStep();
        cout << "Abort at time " << C-> presentTime() << endl;
    }
    else {
        T->validateTimeStep();
        C->validateTimeStep();
    }
} // End loop on timestep size
} // End loop on timesteps

```

4.1.7 End of the coupling calculation

```

T->terminate();
C->terminate();

delete T;
delete C;
closeLib(handle_cathare);
closeLib(handle_trio);
return 0;
}

```

4.2 ICoCo use with a parallel C++ supervisor

To make the comprehension of the writing of a parallel C++ supervisor easier, the writing of the main program will be cut out in small steps. The whole main program can be found in the fourth annex.

Documentation of the Interface for Code Coupling : ICoCo

4.2.1 Useful functions

Due to the fact that we couple two codes and exchange fields between these two codes, some functions can be written to avoid code duplication.

These functions concern:

- To instantiate the problems, if the C++ supervisor is linked to the codes, it has just to call to the *new Problem* methods. If it's not, the supervisor accesses the problems through the *getProblem* function (opens the code dynamic library with *dlopen* method and gets the function with *dlsym* method. This is done in *openLib* function. At the end, remind that the code dynamic library should be closed (*closeLib*).
- To get a one component field from a problem and to return the mean value, the *getFieldMean* function is written. This is necessary because a Cathare cell is connected to a *Trio_U* field on multiple cells
- To set a one component field to a problem with the same value affected to the field cells, the function *setConstantField* is written.
- Some functions to get "min", "max", "sum", "and", and "or" of variables on all processes of *MPI_COMM_WORLD*

```
#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>
#include <sstream>
#include <mpi.h>
```

```
using namespace std;
using namespace ICoCo;
```

```
// Get min, max, and, or of variables on all processes of MPI_COMM_WORLD
```

```
int mpi_min(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
int mpi_max(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
double mpi_min(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
double mpi_max(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
bool mpi_and(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return (result != 0);
}

bool mpi_or(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}

// Get the sum on all processes of comm
// To be called on all processes of comm
double mpi_sum(double d, MPI_Comm comm) {
    double result;
    MPI_Allreduce(&d, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
    return result;
}

// OpenLib function
// Input parameter: the name of the library
// output parameter: a pointer to the problem
// Inout parameter : an opaque "handle" for the dynamic library
Problem* openLib(const char* libname, void* & handle) {
    // open shared library
    Problem *(*getProblem)();
    // open the code dynamic library
    handle =dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
        cerr << dlerror() << endl;
        throw 0;
    }
    // look at getProblem method in the code dynamic library
    getProblem=(Problem* (*)())dlsym(handle, "getProblem");
    if (!getProblem) {
        cout << dlerror() << endl;
        throw 0;
    }
    // instantiation of ICoCo Problem
    return (*getProblem)();
}

// closeLib function
// Input parameter : an opaque "handle" for the dynamic library
void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}
```


Documentation of the Interface for Code Coupling : ICoCo

```

// getFieldMean function
// input parameters: 1- pointer to ICoCo Problem which provides the output
// field
//                      2- name of the field to get
//                      3- MPI_Communicator (equal to 0 if only one process)
// output parameter: the mean of the field (a double)
double getFieldMean(Problem *P, string name, MPI_Comm comm=0) {
    double mean=0.;
    TrioField f ;

    // get the output field from the problem
    P->getOutputField(name,f);
    // only one component fields are treated
    if (f._nb_field_components!=1)
        throw 0;

    // Compute and return the mean value (all elements/nodes have the same
    weight)
    for (int loc=0;loc<f.nb_values();loc++)
        mean=mean+f._field[loc];

    double nb_values=f.nb_values();
    if (comm) {
        mean=mpi_sum(mean,comm);
        nb_values=mpi_sum(nb_values,comm);
    }

    return mean/nb_values;
}

// setConstantField function
// input parameters: 1- pointer to ICoCo Problem to which the field will be
// set
//                      2- name of the field to set
//                      3- value to set
//                      4- component of the field to affect (1st one by default)
// output parameter: No output parameter

void setConstantField(Problem *P, string name, double val, int comp=0) {
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();

    // Fill the values
    // Give a constant field to a Problem.
    // One component of the field is specified, the others are 0.
    int nb_elems=f._nb_elems;
    int nb_comp=f._nb_field_components;
    for (int i=0;i<nb_elems;i++)

```

Documentation of the Interface for Code Coupling : ICoCo

```
for (int j=0;j<nb_comp;j++)
    if (j!=comp)
        f._field[i*nb_comp+j]=0;
    else
        f._field[i*nb_comp+j]=val;

// Give the field to the problem
P->setInputField(name,f);
}
```

4.2.2 MPI initialization

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. MPI_COMM_WORLD is the predefined communicator that includes all of MPI processes. A group is an ordered set of processes.

A new group as a subset of global group and a new communicator are created for Trio_U code. This step allows Trio_U to manage the parallelism by itself inside its group.

The Cathare code will be launched on first processor (rank zero) and Trio_U will be launched on all the others. Therefore, Trio_U group contains all processors except the zero one.

According to the processor number, Cathare Trio_U will be launched.

Different output files are defined to avoid mixing information from the two codes.

```
int main(int argc, char** argv) {

// MPI initialization
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Create the communicators for Cathare and Trio_U
int trio_ids[size-1];
for (int i = 0; i < size - 1; i++)
    trio_ids[i] = i + 1;
int cath_ids[1];
    cath_ids[0] = 0;
MPI_Group group_world;
MPI_Group group_trio;
MPI_Comm comm_trio;
// Create the group world
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
// Create the group trio
MPI_Group_incl(group_world, size - 1, trio_ids, &group_trio);
// Create the communicator trio
MPI_Comm_create(MPI_COMM_WORLD, group_trio, &comm_trio);

// Choose if this process handles Cathare or Trio_U
// the first processor for Cathare, all the others for Trio_U
string progname;
string libname;
if (rank == 0) {
    progname = "Cathare";
    libname = "./libcathare_gad.so";
```

Documentation of the Interface for Code Coupling : ICoCo

```
} else {
  progname = "Trio";
  libname = "_Trio_UModule_opt.so";
}

// Redirect the outputs
ostringstream out;
out << progname << ".out" << ends;
ostringstream err;
err << progname << ".err" << ends;

freopen(out.str().c_str(), "w", stdout);
freopen(err.str().c_str(), "w", stderr);
```

4.2.3 Instantiation of the problems

For each code, a call to function `openLib` described in previous paragraph is enough. If program runs on processor number 0, the problem P represents Cathare. On the other processors, the problem P represents Trio.

```
// Open dynamic libraries
void* handle;
Problem *P=openLib(libname.c_str(), handle);
```

4.2.4 Initialization of the problems

Trio_U needs both data file and a MPI communicator. The communicator is the one created only for Trio_U code defined in §4.2.2

For Cathare code, it is quite different.

In fact, before doing a Cathare calculation process, Cathare needs a pre-processing step where data acquisition is done (description of the hydraulic circuit(s) to be simulated, the events occurring during the simulation and how calculation is managed). This pre-processing step is done and the pre-compiled data file is integrated to the dynamic library loaded at problem instantiation.

Then, the data file has already been defined through the dynamic library loaded at problem instantiation. As a consequence, Cathare doesn't need any initialization parameter.

```
// initialization of the problems
// Cathare doesn't need any initialization parameter
// Trio_U needs both a datafile name and an MPI communicator
if (progname=="Trio") {
  if (size<3)
    P->setDataFile("jdd_Trio.data");
  else
    P->setDataFile("PAR_jdd_Trio.data");
  P->setMPIComm(&comm_trio);
}
P->initialize();
```

Documentation of the Interface for Code Coupling : ICoCo

4.2.5 Initialization of coupling parameters and data to transform data

There is in this part no difference with the sequential mode.

```
// Coupling parameters
// Time for which Cathare runs alone. Allows to reach a Cathare steady
state.
double t_begin_trio=1000;
// Time at which Trio_U -> Cathare retroaction on pressure begins
double time_retro_P=1001;
// Time at which Trio_U -> Cathare retroaction on temperature begins
double time_retro_T=1003;

// data to transform Cathare enthalpy into Trio_U temperature or vice versa
// H = href + Cp_Trio * ( T - Tref )
double href = 1.15116e6 ;
double tref = 263.8+273.15 ; // Trio_U temperature are in K
double Cp_Trio = 5332.43 ;

// mass flowrate transformation
// In Cathare, pipe of diameter 0.6m (S=0.28274m²)
// In Trio_U, 2D pipe of section 0.6m
// We keep the same Q=rho*u*S between Cathare and Trio_U
// => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
// !! u(Trio) != u(Cathare)
double S_Trio = 0.6 ;
double rho_Trio = 739.206 ;

// variables for pressure retroaction
// Must be remembered from timestep to timestep
double dP1=0; // outlet1 - inlet1
double dP2=0; // outlet2 - inlet1
double dP3=0; // inlet2 - inlet1
```

4.2.6 First time loop : Cathare alone to reach a Cathare steady state

During the first 1000 seconds, Cathare will run alone to reach a Cathare steady state.

```
// First time loop (Cathare alone)
if (programe=="Cathare") {
  double epsilon=1e-10; // small time for double comparisons
  bool stop; // Does the problem want to stop ?

  while (1) { // Loop on timesteps
    double present=P->presentTime();
    double dt=P->computeTimeStep(stop);
    if (stop || present>t_begin_trio-epsilon)
      break;

    // Modify dt in order to come to exactly t_begin_trio
    if (present+dt>t_begin_trio)
      dt=t_begin_trio-present;
    else if (present+2*dt>t_begin_trio)
      dt=(t_begin_trio-present)/2;
```

Documentation of the Interface for Code Coupling : ICoCo

```

// Initialize the timestep
P->initTimeStep(dt);

// Perform the computation
bool ok=P->solveTimeStep();

// Either validate or abort it
if (!ok) // The resolution failed, try with a new dt
    P->abortTimeStep();
else // Validate and go to the next time step
    P->validateTimeStep();
} // End loop on timesteps
}

```

4.2.7 Second time loop : both codes with one-way coupling

This part contains the same than in sequential mode with moreover the data transfer between the different processors (MPI_Bcast). MPI_Bcast sends a variable value from one process to all other processes of the group. If the process number is zero, it means that it is a Cathare value which is sent to other processes. If the process number is one, it means that it is a Trio_U value which is sent to all other processes.

The boolean values, to determine if the final time has been reached or to know is the time step calculation has correctly been performed, are calculated with the mpi_and and mpi_or functions described in §4.2.1.

```

// Second time loop (both codes)
bool ok=true; // Is the time interval successfully solved?
bool stop=false;

while (!stop) { // Loop on timesteps
    ok=false;
    while (!ok) { // Loop on timestep size
        // Compute time step length
        double dt =P->computeTimeStep(stop);
        dt=mpi_min(dt);
        // And decide if we have to stop
        stop = mpi_or(stop);
        if (stop)
            break;

        //prepare the new time step
        P->initTimeStep(dt);

        // Get output from Cathare
        double C_pressure_inlet1, C_pressure_outlet1;
        double C_pressure_inlet2, C_pressure_outlet2;
        double C_massflow_inlet1, C_massflow_outlet1;
        double C_massflow_inlet2, C_massflow_outlet2;
        double C_enthalpy_inlet1, C_enthalpy_outlet1;
        double C_enthalpy_inlet2, C_enthalpy_outlet2;
    }
}

```

Documentation of the Interface for Code Coupling : ICoCo

```
if (programe=="Cathare") {
  // Pressures
  C_pressure_inlet1=getFieldMean(P,"PRESSURE_PIPE11_85");
  C_pressure_outlet1=getFieldMean(P,"PRESSURE_PIPE12_1");
  C_pressure_inlet2=getFieldMean(P,"PRESSURE_PIPE21_85");
  C_pressure_outlet2=getFieldMean(P,"PRESSURE_PIPE22_1");

  // Mass flow rates
  C_massflow_inlet1=getFieldMean(P,"LIQFLOW_PIPE11_85");
  C_massflow_outlet1=getFieldMean(P,"LIQFLOW_PIPE12_1");
  C_massflow_inlet2=getFieldMean(P,"LIQFLOW_PIPE21_85");
  C_massflow_outlet2=getFieldMean(P,"LIQFLOW_PIPE22_1");

  // Enthalpies
  C_enthalpy_inlet1=getFieldMean(P,"LIQH_PIPE11_85");
  C_enthalpy_outlet1=getFieldMean(P,"LIQH_PIPE12_1");
  C_enthalpy_inlet2=getFieldMean(P,"LIQH_PIPE21_85");
  C_enthalpy_outlet2=getFieldMean(P,"LIQH_PIPE22_1");
}

// Transfer the data to Trio_U
MPI_Bcast(&C_pressure_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Bcast(&C_massflow_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Compute Trio_U input values
//calculation of input temperature for Trio_U
double temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref;
double temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref;
double temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref;
double temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref;

// calculation of inlet and outlet velocity for Trio_U
double u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio;
double u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio;
double u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio;
double u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio;

// Modifications to ensure div(u)=0
// works because all areas are equal.
double divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2;
u_outlet1-=divergence/4;
```

Documentation of the Interface for Code Coupling : ICoCo

```
u_outlet2-=divergence/4;
u_inlet1+=divergence/4;
u_inlet2+=divergence/4;
```

```
// Gets output from Trio_U
```

```
double T_pressure_inlet1, T_pressure_inlet2 ;
double T_pressure_outlet1, T_pressure_outlet2;
double T_temperature_inlet1, T_temperature_inlet2 ;
double T_temperature_outlet1, T_temperature_outlet2;
```

```
if (programe=="Trio") {
    // Pressures
```

```
T_pressure_inlet1=getFieldMean(P,"pressure_inlet1",comm_trio)*rho_Trio;
T_pressure_inlet2=getFieldMean(P,"pressure_inlet2",comm_trio)*rho_Trio;
T_pressure_outlet1=getFieldMean(P,"pressure_outlet1",comm_trio)*rho_Trio;
T_pressure_outlet2=getFieldMean(P,"pressure_outlet2",comm_trio)*rho_Trio;
```

```
    // Temperatures
```

```
    T_temperature_inlet1=getFieldMean(P,"temperature_inlet1",comm_trio);
    T_temperature_inlet2=getFieldMean(P,"temperature_inlet2",comm_trio);
```

```
T_temperature_outlet1=getFieldMean(P,"temperature_outlet1",comm_trio);
```

```
T_temperature_outlet2=getFieldMean(P,"temperature_outlet2",comm_trio);
}
```

```
// Transfer the data to Cathare
```

```
MPI_Bcast(&T_pressure_inlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_outlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_inlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_outlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
```

```
MPI_Bcast(&T_temperature_inlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_inlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_outlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_outlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
```

```
// Give input to Cathare
```

```
// Trio_U -> Cathare retroaction on pressure
```

```
if (programe=="Cathare") {
    if (P->presentTime()>time_retro_P) {
        double dP_T1 = T_pressure_inlet1-T_pressure_outlet1;
        double dP_T2 = T_pressure_inlet1-T_pressure_outlet2;
        double dP_T3 = T_pressure_inlet1-T_pressure_inlet2;

        double dP_C1 = C_pressure_inlet1-C_pressure_outlet1;
        double dP_C2 = C_pressure_inlet1-C_pressure_outlet2;
        double dP_C3 = C_pressure_inlet1-C_pressure_inlet2;
```

Documentation of the Interface for Code Coupling : ICoCo

```

double tau_coupl=0.1;
double k_loc=0;
if (dt>tau_coupl)
    k_loc=1;
else
    k_loc=dt/tau_coupl;

// pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc;
dP2 += (dP_C2-dP_T2)*k_loc;
dP3 -= (dP_C3-dP_T3)*k_loc;

setConstantField(P,"DPLEXT_PIPE12_1",dP1);
setConstantField(P,"DPLEXT_PIPE22_1",dP2);
setConstantField(P,"DPLEXT_PIPE21_86",dP3);
}

// Trio_U -> Cathare retroaction on temperature
if (P->presentTime()>time_retro_T) {
double T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref);
double T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref);
double T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-
tref);
double T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-
tref);

// mesh numbers are growing from "left" to "right"
// overlapping domain on the "right"
// Loop 1
setConstantField(P,"OVHLEXT_PIPE11_85",-10);
setConstantField(P,"OVPLEXT_PIPE11_85",-10);
setConstantField(P,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1);
// Loop 2
setConstantField(P,"OVHLEXT_PIPE21_85",-10);
setConstantField(P,"OVPLEXT_PIPE21_85",-10);
setConstantField(P,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2);

// overlapping domain on the "left"
// Loop 1
setConstantField(P,"OVHLEXT_PIPE12_1",10);
setConstantField(P,"OVPLEXT_PIPE12_2",10);
setConstantField(P,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1);
// Loop 2
setConstantField(P,"OVHLEXT_PIPE22_1",10);
setConstantField(P,"OVPLEXT_PIPE22_2",10);
setConstantField(P,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2);
}
}
// Give input to Trio_U
if (programe=="Trio") {
// Velocities
setConstantField(P,"u_inlet1",u_inlet1,1);
setConstantField(P,"u_inlet2",u_inlet2,1);

```


Documentation of the Interface for Code Coupling : ICoCo

```
setConstantField(P,"u_outlet1",u_outlet1,1);
setConstantField(P,"u_outlet2",u_outlet2,1);

// Temperatures
setConstantField(P,"temperature_inlet1",temperature_inlet1);
setConstantField(P,"temperature_inlet2",temperature_inlet2);
setConstantField(P,"temperature_outlet1",temperature_outlet1);
setConstantField(P,"temperature_outlet2",temperature_outlet2);
}
// Solve next time step
bool ok =P->solveTimeStep();
ok =mpi_and(ok) ;

// Two ways : The resolution failed, try with a new dt or validate and
go
// to next time step
if (!ok) {
    P->abortTimeStep();
    cout << "Abort at time " << P-> presentTime() << endl;
}
else {
    P->validateTimeStep();
}
} // End loop on timestep size
} // End loop on timesteps
```

4.2.8 End of the coupling calculation

In comparison with the sequential mode, one just has to synchronize all processors and to terminate the MPI execution environment.

```
P->terminate();

delete P;
closeLib(handle);
MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();

return 0;
}
```

4.3 I CoCo use with a sequential python supervisor

Thanks to swig software development tool, it is easy to connect program written in C++ with Python. In fact, to keep the possibility of using the functions described in §4.1.1, they are always written in C++. Then, a header file with the declaration of the 4 functions and a source file are used.

4.3.1 Useful functions

The header file is listed below.

```
#ifndef main_seq_inclus
#define main_seq_inclus
```

Documentation of the Interface for Code Coupling : ICoCo

```
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>
namespace ICoCo {
class Problem;
}
using namespace std;
using namespace ICoCo;

// OpenLib function
// Input parameter: the name of the library
// output parameter: a pointer to the problem
// Inout parameter : an opaque "handle" for the dynamic library
Problem* openLib(const char* libname, void* &handle) ;

// closeLib function
// Input parameter: an opaque "handle" for the dynamic library
void closeLib(void* handle);

// getFieldMean function
// input parameters: 1- pointer to ICoCo Problem which provides the output
field
//
//           2- name of the field to get
// output parameter: the mean of the field (a double)
double getFieldMean(Problem *P, const char*) ;

// setConstantField function
// input parameters: 1- pointer to ICoCo Problem to which the field will be
set
//
//           2- name of the field to set
//           3- value to set
//           4- component of the field to affect (1st one by default)
// output parameter: No output parameter
void setConstantField(Problem *P, const char*, double val, int comp=0) ;

#endif
```

The source file is listed below:

```
#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>

using namespace std;
using namespace ICoCo;

// Open a shared library
// Find the function getProblem
// Run it and return a pointer to the Problem
Problem* openLib(const char* libname, void* & handle) {
    Problem *(*getProblem)();
    handle=dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
```

Documentation of the Interface for Code Coupling : ICoCo

```

    cerr << dlerror() << endl;
    throw 0;
}
getProblem=(Problem* (*)())dlsym(handle, "getProblem");
if (!getProblem) {
    cout << dlerror() << endl;
    throw 0;
}
return (*getProblem)();
}

// Close a shared library
// Prints the errors if any

void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

// Get a field from a Problem (1 component)
// And returns the mean value
// (all elems/nodes have the same weight!)
double getFieldMean(Problem *P, const char* name) {
    double mean=0;
    TrioField f;

    // Get the output field
    P->getOutputField(name,f);
    if (f._nb_field_components!=1)
        throw 0;
    // Compute and return the mean value
    for (int loc=0;loc<f.nb_values();loc++)
        mean=mean+f._field[loc];
    return mean/f.nb_values();
}

// Give a constant field to a Problem.
// One component of the field is specified, the others are 0.
void setConstantField(Problem *P, const char* name , double val, int comp=0)
{
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();
    // Fill the values
    int nb_elems=f._nb_elems;
    int nb_comp=f._nb_field_components;
    for (int i=0;i<nb_elems;i++)

```

Documentation of the Interface for Code Coupling : ICoCo

```

for (int j=0;j<nb_comp;j++)
    if (j!=comp)
        f._field[i*nb_comp+j]=0;
    else
        f._field[i*nb_comp+j]=val;

// Give the field to the problem
P->setInputField(name,f);
}

```

4.3.2 Swig interface file

The interface file is the input to Swig. An interface file contains all header files or functions declarations that will be used in python. The module name is the one that will be import after in python script. Everything in the `%{...}` block is simply copied to the resulting wrapper file created by swig. This section is used to include header files and other declarations that are required to make the generated wrapper code compile.

Then, one has to indicate to swig the library file it will need for strings.

Finally, the include files that have to be analyzed by swig are listed. The header file "main_seq.h" is a copy of the one containing the useful functions described in §4.3.1 but without the declaration of `openLib` and `closeLib` functions. These 2 functions created some problems to swig interpreter because of the handle. We will redefine these two functions in swig interface file.

Note moreover that for the ICoCo header, the name of the file is different. In fact, it is a copy of *Problem.h* header without the declaration of the `getProblem` function which creates undefined symbol problems with swig and that is not used in python (the `getProblem` function will be called through C++ `openLib` function).

The `computeTimeStep` function of `ICoCo::Problem` class is also redefined because of the output boolean pointer argument. This is made to avoid the use of `cpointer` swig tool.

To resume, the swig interface file redefines three functions:

- `openLib` : this new function takes only one input parameter and returns a tuple containing the problem and the handle
- `closeLib` : this new function is redefined because of the wrapping of the handle made in `openLib` function
- `computeTimeStep` : this new function has no input parameter and returns a tuple containing the timestep and the boolean

```

%module ICoCoModule
%{
#include "main_seq.h"
#include <Problem.h>
%}
#include "std_string.i"
#include "Problem.hxx"
#include "main_seq.hxx"

// The following code gets inserted into the result python file:
// create needed python symbols
%inline %{
PyObject *openLib(const char *libname)
{

```

Documentation of the Interface for Code Coupling : ICoCo

```

    cout << "libname" << libname << endl;
    void *outParam;
    Problem *ret0=openLib(libname,outParam);
    PyObject *ret=PyTuple_New(2);

PyTuple_SetItem(ret,0,SWIG_NewPointerObj(SWIG_as_voidptr(ret0),SWIGTYPE_p_ICo
Co__Problem, 0 | 0 ));

PyTuple_SetItem(ret,1,SWIG_NewPointerObj(SWIG_as_voidptr(outParam),SWIGTYPE_p
_ICoCo__Problem, 0 | 0 ));
    return ret;
}

void closeLib(PyObject*obj)
{
    void *argp;
    int status=SWIG_ConvertPtr(obj,&argp,SWIGTYPE_p_ICoCo__Problem,0|0);
    closeLib(argp);
}
%}

%extend ICoCo::Problem {
    PyObject* computeTimeStep()
    {
        bool stop;
        double dt;
        dt = (*self).computeTimeStep(stop);
        cout << "double dt" << dt << endl;
        cout << "stop " << stop << endl;
        PyObject *ret=PyTuple_New(2);
        PyTuple_SetItem(ret,0,PyFloat_FromDouble(dt));
        PyTuple_SetItem(ret,1,PyBool_FromLong(stop));
        return ret;
    }
}

```

After a compilation step, an extension module `_ICoCoModule.so` is created. It can now be used in a python script.

The following paragraphs will detail the python supervisor to launch the use case described in §3. The complete python script can be found in fifth annex.

4.3.3 Instantiation of the problems

First, we need to import `ICoCoModule`, created by swig tool and particularly the useful functions. For each code, a call to function `openLib` is enough.

```

import ICoCoModule,os
from ICoCoModule import getFieldMean,setConstantField,openLib, closeLib

# open shared libraries
T, handle_Trio=openLib("_Trio_UModule_opt.so")
C, handle_Cathare=openLib("./libcathare_gad.so")

```

Documentation of the Interface for Code Coupling : ICoCo

4.3.4 Initialization of the problems

Trio_U needs a data file.

For Cathare code, it is quite different.

In fact, before doing a Cathare calculation process, Cathare needs a pre-processing step where data acquisition is done (description of the hydraulic circuit(s) to be simulated, the events occurring during the simulation and how calculation is managed). This pre-processing step is done and the pre-compiled data file is integrated to the dynamic library loaded at problem instantiation.

Then, the data file has already been defined through the dynamic library loaded at problem instantiation. As a consequence, Cathare doesn't need any initialization parameter.

```
# Initialize the problems

T.setDataFile("jdd_Trio.data")
T.initialize()

C.initialize()
```

4.3.5 Initialization of coupling parameters and data to transform data

This paragraph contains exactly the same as for a C++ sequential supervisor.

```
# Coupling parameters
# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
# H = href + Cp_Trio * ( T - Tref )
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m (S=0.28274m²)
# In Trio_U, 2D pipe of section 0.6m
# We keep the same Q=rho*u*S between Cathare and Trio_U
# => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
# !! u(Trio) != u(Cathare)
S_Trio = 0.6
rho_Trio = 739.206

# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1
```

Documentation of the Interface for Code Coupling : ICoCo

4.3.6 First time loop : Cathare alone to reach a Cathare steady state

During the first 1000 seconds, Cathare will run alone to reach a Cathare steady state.

First time loop (Cathare alone)

```
epsilon=1e-10 # small time for double comparisons

while (1) : # Loop on timesteps
  present=C.presentTime()
  dt, stop =C.computeTimeStep()
  if (stop or present>t_begin_trio-epsilon):
    break

  # Modify dt in order to come to exactly t_begin_trio
  if (present+dt>t_begin_trio):
    dt=t_begin_trio-present
  elif (present+2*dt>t_begin_trio):
    dt=(t_begin_trio-present)/2

  # Initialize the timestep
  C.initTimeStep(dt)

  # Perform the computation
  ok=C.solveTimeStep()

  # Either validates or aborts it
  if not(ok): # The resolution failed, try with a new dt
    C.abortTimeStep()
  else: # Validate and go to the next time step
    C.validateTimeStep()
  pass # End loop on timesteps
```

4.3.7 Second time loop : both codes with one-way coupling

This part contains computation of the coupling time step, initialization of the time step for the two codes, getting output from Cathare, and Trio_U, giving input to Trio_U and Cathare, launching the codes and go away if everything OK until final time step has been reached.

Following data are calculated by Cathare and given to Trio_U:

- The mass flow at each boundary between the two codes comes from Cathare
- Cathare gives the enthalpies at the boundaries, they are converted into temperatures which are given to Trio_U. They are used only at the boundaries where the gas flows towards the flow domain

These are data given by Trio_U to Cathare: the feedbacks :

- Trio_U gives the temperatures at the boundaries, they are converted into enthalpies which are given to Cathare. They are used only at the boundaries where the gas flow outwards the Trio_U domain.
- The pressure at one of the Trio_U boundaries is taken as the reference. At the other boundaries, the pressure drop between the reference pressure and the pressure at the boundary is calculated and imposed to Cathare

Documentation of the Interface for Code Coupling : ICoCo

```
# Second time loop (both codes)
ok=True      # Is the time interval successfully solved?
stop=False

while not(stop) :                               # Loop on timesteps
    ok=False
    while not(ok) :                               # Loop on timestep size
        # Compute time step length
        dt_C, stop_C=C.computeTimeStep()
        dt_T, stop_T=T.computeTimeStep()
        dt=min(dt_C,dt_T)

        # And decide if we have to stop
        stop = stop_T or stop_C
        if (stop):
            break

        # prepare the new time step
        C.initTimeStep(dt)
        T.initTimeStep(dt)

        # Get output from Cathare
        # Pressures
        C_pressure_inlet1=getFieldMean(C,"PRESSURE_PIPE11_85")
        C_pressure_outlet1=getFieldMean(C,"PRESSURE_PIPE12_1")
        C_pressure_inlet2=getFieldMean(C,"PRESSURE_PIPE21_85")
        C_pressure_outlet2=getFieldMean(C,"PRESSURE_PIPE22_1")

        # Mass flow rates
        C_massflow_inlet1=getFieldMean(C,"LIQFLOW_PIPE11_85")
        C_massflow_outlet1=getFieldMean(C,"LIQFLOW_PIPE12_1")
        C_massflow_inlet2=getFieldMean(C,"LIQFLOW_PIPE21_85")
        C_massflow_outlet2=getFieldMean(C,"LIQFLOW_PIPE22_1")
        # Enthalpies
        C_enthalpy_inlet1=getFieldMean(C,"LIQH_PIPE11_85")
        C_enthalpy_outlet1=getFieldMean(C,"LIQH_PIPE12_1")
        C_enthalpy_inlet2=getFieldMean(C,"LIQH_PIPE21_85")
        C_enthalpy_outlet2=getFieldMean(C,"LIQH_PIPE22_1")

        # Compute Trio_U input values
        #calculation of input temperature for Trio_U
        temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
        temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
        temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
        temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref

        # calculation of inlet and outlet velocity for Trio_U
        u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
        u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
        u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
        u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio

        # Modifications to ensure div(u)=0
```


Documentation of the Interface for Code Coupling : ICoCo

```
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4

# Get output from Trio_U
# Pressures
T_pressure_inlet1=getFieldMean(T,"pressure_inlet1")*rho_Trio
T_pressure_inlet2=getFieldMean(T,"pressure_inlet2")*rho_Trio
T_pressure_outlet1=getFieldMean(T,"pressure_outlet1")*rho_Trio
T_pressure_outlet2=getFieldMean(T,"pressure_outlet2")*rho_Trio

# Temperatures
T_temperature_inlet1=getFieldMean(T,"temperature_inlet1")
T_temperature_inlet2=getFieldMean(T,"temperature_inlet2")
T_temperature_outlet1=getFieldMean(T,"temperature_outlet1")
T_temperature_outlet2=getFieldMean(T,"temperature_outlet2")

# Give input to Cathare
# Trio_U -> Cathare retroaction on pressure
if (C.presentTime(>time_retro_P) :
    dP_T1 = T_pressure_inlet1-T_pressure_outlet1
    dP_T2 = T_pressure_inlet1-T_pressure_outlet2
    dP_T3 = T_pressure_inlet1-T_pressure_inlet2

    dP_C1 = C_pressure_inlet1-C_pressure_outlet1
    dP_C2 = C_pressure_inlet1-C_pressure_outlet2
    dP_C3 = C_pressure_inlet1-C_pressure_inlet2

    tau_coupl=0.1
    k_loc=0
    if (dt>tau_coupl):
        k_loc=1
    else:
        k_loc=dt/tau_coupl

# pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc
dP2 += (dP_C2-dP_T2)*k_loc
dP3 -= (dP_C3-dP_T3)*k_loc

setConstantField(C,"DPLEXT_PIPE12_1",dP1)
setConstantField(C,"DPLEXT_PIPE22_1",dP2)
setConstantField(C,"DPLEXT_PIPE21_86",dP3)
pass

# Trio_U -> Cathare retroaction on temperature
if (C.presentTime(>time_retro_T) :
    T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
    T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
    T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
```

Documentation of the Interface for Code Coupling : ICoCo

```
T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)
```

```
# mesh numbers are growing from "left" to "right"
```

```
# overlapping domain on the "right"
```

```
# Loop 1
```

```
setConstantField(C,"OVHLEXT_PIPE11_85",-10)
```

```
setConstantField(C,"OVPLEXT_PIPE11_85",-10)
```

```
setConstantField(C,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1)
```

```
# Loop 2
```

```
setConstantField(C,"OVHLEXT_PIPE21_85",-10)
```

```
setConstantField(C,"OVPLEXT_PIPE21_85",-10)
```

```
setConstantField(C,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2)
```

```
# overlapping domain on the "left"
```

```
# Loop 1
```

```
setConstantField(C,"OVHLEXT_PIPE12_1",10)
```

```
setConstantField(C,"OVPLEXT_PIPE12_2",10)
```

```
setConstantField(C,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1)
```

```
# Loop 2
```

```
setConstantField(C,"OVHLEXT_PIPE22_1",10)
```

```
setConstantField(C,"OVPLEXT_PIPE22_2",10)
```

```
setConstantField(C,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2)
```

```
pass
```

```
# Give input to Trio_U
```

```
# Velocities
```

```
setConstantField(T,"u_inlet1",u_inlet1,1)
```

```
setConstantField(T,"u_inlet2",u_inlet2,1)
```

```
setConstantField(T,"u_outlet1",u_outlet1,1)
```

```
setConstantField(T,"u_outlet2",u_outlet2,1)
```

```
# Temperatures
```

```
setConstantField(T,"temperature_inlet1",temperature_inlet1)
```

```
setConstantField(T,"temperature_inlet2",temperature_inlet2)
```

```
setConstantField(T,"temperature_outlet1",temperature_outlet1)
```

```
setConstantField(T,"temperature_outlet2",temperature_outlet2)
```

```
# Solve next time step
```

```
ok_T=T.solveTimeStep()
```

```
ok_C=C.solveTimeStep()
```

```
ok = ok_T and ok_C
```

```
# Two ways : The resolution failed, try with a new dt or validate and go
```

```
# to next time step
```

```
if not(ok) :
```

```
    T.abortTimeStep()
```

```
    C.abortTimeStep()
```

```
    print "Abort at time " + C.presentTime()
```

```
else:
```

```
    T.validateTimeStep()
```

```
    C.validateTimeStep()
```

```
pass
```

```
# End loop on timestep size
```

```
pass
```

```
# End loop on timesteps
```

4.3.8 End of the coupling calculation

```
T.terminate()  
C.terminate()  
closeLib(handle_Trio)  
closeLib(handle_Cathare)
```

4.4 ICoCo use with a parallel python supervisor

Thanks to swig software development tool, it is easy to connect program written in C++ with Python. In fact, to keep the possibility of using the functions described in §4.1.1, they are always written in C++. Then, a header file with the declaration of the 4 useful functions and some other MPI functions (*“min”*, *“max”*, *“sum”*, *“and”*, and *“or”*, *“broadcast”*) and a source file are used.

There are two solutions to launch a parallel python supervisor:

- Initialize MPI in the python script. This means that one has to use an implementation of MPI for Python (like Scientific MPI).
- Initialize MPI in the C++ part. This means that one has to use an implementation of MPI for C++.

In the case of the Cathare/Trio_U coupling, there is already OpenMPI which is used by Trio_U. Then, one decides to perform the second solution. But, mind that OpenMPI furnished by Trio_U has to be compiled enabling the generation of dynamic libraries.

This choice involves defining some MPI initialization functions in the C++ part (header and source file).

These functions are:

- `my_MPI_Init`: to initialize MPI,
- `my_MPI_Comm_size`: to determine the size of the group associated with `MPI_COMM_WORLD` communicator
- `my_MPI_Comm_rank`: to determine the rank of the calling process in the `MPI_COMM_WORLD` communicator
- `MPI_Comm_new`: to create a new communicator associated with a smaller group with process indicated in input parameter

4.4.1 Useful functions

The header file is listed below.

```
#ifndef main_para_inclus  
#define main_para_inclus  
  
#include <ICoCoTrioField.h>  
#include <dlfcn.h>  
#include <iostream>  
#include <mpi.h>  
  
namespace ICoCo {  
class Problem;  
}  
using namespace std;  
using namespace ICoCo;  
  
void my_MPI_Init(int* argc, char*** argv);  
int my_MPI_Comm_size();  
int my_MPI_Comm_rank();  
// MPI_Comm_new
```

Documentation of the Interface for Code Coupling : ICoCo

```

// create a new group corresponding to world communicator
// create a smaller group
// create communicator corresponding to smaller group
MPI_Comm* MPI_Comm_new(int size,int* trio_ids);

// Get min, max, and, or of variables on all processes of MPI_COMM_WORLD
int mpi_min(int i) ;
int mpi_max(int i);
double mpi_min(double i);
double mpi_max(double i) ;
bool mpi_and(bool b) ;
bool mpi_or(bool b) ;
// broadcast d from root to all other process
double mpi_bcast(double d, int root);

// OpenLib function
// Open a shared library
// Find the function getProblem
// Run it and return a pointer to the Problem
Problem* openLib(const char* libname, void* & handle) ;

// closeLib function
// Close a shared library
// Prints the errors if any
void closeLib(void* handle) ;

// getFieldMean function
// Get a field from a Problem (1 component)
// And returns the mean value
// (all elems/nodes have the same weight!)
double getFieldMean(Problem *P, const char*,MPI_Comm* comm=0) ;

// setConstantField function
// Give a constant field to a Problem.
// One component of the field is specified, the others are 0.
void setConstantField(Problem *P, const char*, double val, int comp=0) ;

#endif

```

The source file is listed below:

```

#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>
#include <fstream>
#include <vector>
#include "mpi.h"
#include <assert.h>

using namespace std;
using namespace ICoCo;

// MPI_Comm_Init

```

Documentation of the Interface for Code Coupling : ICoCo

```
void my_MPI_Init(int* argc,char*** argv) {
    int flag;
    MPI_Initialized(&flag);
    // cerr<<" MPI initialized dans SWIG " <<flag<<endl;
    if (flag==0) {
        if (MPI_Init(argc,argv)!=MPI_SUCCESS) cerr<<"Problem with MPI_Init
"<<endl ;
    }
}

// =====
// MPI_Comm_size
// =====
int my_MPI_Comm_size() {
    int res = 0;
    int err = MPI_Comm_size(MPI_COMM_WORLD, &res);
    if ( err != MPI_SUCCESS )
        cerr<<"Problem with MPI_Comm_size "<<endl ;
    return res ;
}

// =====
// MPI_Comm_rank
// =====
int my_MPI_Comm_rank() {
    int res = 0;
    int err = MPI_Comm_rank(MPI_COMM_WORLD, &res);
    if ( err != MPI_SUCCESS )
        cerr<<"Problem with MPI_Comm_rank "<<endl ;
    return res;
}

// =====
// MPI_Comm_new
// create a new group corresponding to world communicator
// create a smaller group
// create communicator corresponding to smaller group
// =====
MPI_Comm* MPI_Comm_new(int size,int* trio_ids) {

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Group group_world;
    MPI_Group group_trio;
    MPI_Comm* comm_trio;
    comm_trio=(MPI_Comm*)malloc(sizeof(MPI_Comm));
    MPI_Comm_group(MPI_COMM_WORLD, &group_world); // Create the group world

    MPI_Group_incl(group_world, size , trio_ids, &group_trio); // Create the
group trio
    MPI_Comm_create(MPI_COMM_WORLD, group_trio, comm_trio);

    return comm_trio;
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
}

// Get min, max, and, or of variables on all processes of MPI_COMM_WORLD
int mpi_min(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
int mpi_max(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
double mpi_min(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
double mpi_max(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
bool mpi_and(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return (result != 0);
}
bool mpi_or(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
double mpi_sum(double d, MPI_Comm comm) {
    double result;
    MPI_Allreduce(&d, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
    return result;
}
double mpi_bcast(double d, int root) {
    MPI_Bcast(&d, 1, MPI_DOUBLE, root, MPI_COMM_WORLD);
    return d;
}

// Open a shared library
// Find the function getProblem
// Run it and return a pointer to the Problem
Problem* openLib(const char* libname, void* & handle) {
    Problem *(*getProblem)();
    handle=dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
```

Documentation of the Interface for Code Coupling : ICoCo

```

    cerr << dlerror() << endl;
    throw 0;
}
getProblem=(Problem* (*)())dlsym(handle, "getProblem");
if (!getProblem) {
    cout << dlerror() << endl;
    throw 0;
}
return (*getProblem)();
}

// Close a shared library
// Prints the errors if any
void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

// Get a field from a Problem (1 component)
// And returns the mean value (all elems/nodes have the same weight!)
double getFieldMean(Problem *P, const char* name, MPI_Comm* comm=0) {
    double mean=0;
    TrioField f;
    // Get the output field
    P->getOutputField(name,f);
    if (f._nb_field_components!=1)
        throw 0;

    // Compute and return the mean value
    for (int loc=0;loc<f.nb_values();loc++)
        mean=mean+f._field[loc];
    double nb_values=f.nb_values();
    if (comm) {
        mean=mpi_sum(mean,*comm);
        nb_values=mpi_sum(nb_values,*comm);
    }
    return mean/nb_values;
}

// Give a constant field to a Problem.
// One component of the field is specified, the others are 0.
void setConstantField(Problem *P, const char* name , double val, int comp=0)
{
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();

    // Fill the values

```

Documentation of the Interface for Code Coupling : ICoCo

```
int nb_elems=f._nb_elems;
int nb_comp=f._nb_field_components;
for (int i=0;i<nb_elems;i++)
  for (int j=0;j<nb_comp;j++)
    if (j!=comp)
      f._field[i*nb_comp+j]=0;
    else
      f._field[i*nb_comp+j]=val;

// Give the field to the problem
P->setInputField(name,f);
}
```

4.4.2 Swig interface file

The interface file is the input to Swig. An interface file contains all header files or functions declarations that will be used in python. The module name is the one that will be import after in python script. Everything in the `%{...}` block is simply copied to the resulting wrapper file created by swig. This section is used to include header files and other declarations that are required to make the generated wrapper code compile.

Then, one has to indicate to swig the library file it will need for strings.

Finally, the include files that have to be analyzed by swig are listed. The header file "main_para.h" is a copy of the one containing the useful functions described in §4.4.1 but without the declaration of `openLib` and `closeLib` functions. These 2 functions created some problems to swig interpreter because of the handle. We will redefine these two functions in swig interface file.

Note moreover that for the ICoCo header, the name of the file is different. In fact, it is a copy of *Problem.h* header without the declaration of the `getProblem` function which creates undefined symbol problems with swig and that is not used in python (the `getProblem` function will be called through C++ `openLib` function).

The `computeTimeStep` function of `ICoCo::Problem` class is also redefined because of the output boolean pointer argument. This is made to avoid the use of `cpointer` swig tool.

To resume, the swig interface file redefines following functions:

- `openLib` : this new function takes only one input parameter and returns a tuple containing the problem and the handle
- `closeLib` : this new function is redefined because of the wrapping of the handle made in `openLib` function
- `computeTimeStep` : this new function has no input parameter and returns a tuple containing the timestep and the boolean

In the swig interface file, some new "typemaps" are redefined. The typemaps are used to define conversion of datatypes between C++ and python. In this case, two new conversions are described: one for `argc,argv` for MPI initialization and the second one for a list of int (a list of int is used to describe process devoted to `Trio_U`).

```
%module ICoCoModule
%{
#include "main_para.h"

#include <Problem.h>
#include <mpi.h>
%}
#include "std_string.i"
```


Documentation of the Interface for Code Coupling : ICoCo

```
// Creates "int *argc, char ***argv" parameters from input list
%typemap(in) (int *argc, char ***argv) {
    int i;
    if (!PyList_Check($input)) {
        PyErr_SetString(PyExc_ValueError, "Expecting a list");
        return NULL;
    }
    int aSize = PyList_Size($input);
    $1 = &aSize;
    char** aStrs = (char **) malloc((aSize+1)*sizeof(char *));
    for (i = 0; i < aSize; i++) {
        PyObject *s = PyList_GetItem($input,i);
        if (!PyString_Check(s)) {
            free(aStrs);
            PyErr_SetString(PyExc_ValueError, "List items must be strings");
            return NULL;
        }
        aStrs[i] = PyString_AsString(s);
    }
    aStrs[i] = 0;
    $2 = &aStrs;
}

%typemap(freearg) (int *argc, char ***argv) {
    if ($2) free(*($2));
}

// Multi-argument typemap, here for a list of integers.
%typemap(in) int * {
    if (PyList_Check($input)) {
        int size = PyList_Size($input);
        int i = 0;
        $1 = (int *) malloc(size*sizeof(int));
        for (i = 0; i < size; i++) {
            PyObject *o = PyList_GetItem($input,i);
            if (PyInt_Check(o)) {
                $1[i] = PyLong_AsLong(PyList_GetItem($input,i));
                cout << "$1[" << i << "] = " << $1[i] << endl;
            }
        }
        else {
            PyErr_SetString(PyExc_TypeError,"list must contain int");
            free($1);
            return NULL;
        }
    }
}
else {
    PyErr_SetString(PyExc_TypeError,"not a list");
    return NULL;
}
}
```

Documentation of the Interface for Code Coupling : ICoCo

```

%typemap(freearg) int * {
    if ($1) free($1);
}

#include "Problem.hxx"
#include "main_para.hxx"

// The following code gets inserted into the result python file:
// create needed python symbols
%inline %{
    PyObject *openLib(const char *libname)
    {
        cout << "libname" << libname << endl;
        void *outParam;
        Problem *ret0=openLib(libname,outParam);
        PyObject *ret=PyTuple_New(2);

PyTuple_SetItem(ret,0,SWIG_NewPointerObj(SWIG_as_voidptr(ret0),SWIGTYPE_p_ICo
Co__Problem, 0 | 0 ));

PyTuple_SetItem(ret,1,SWIG_NewPointerObj(SWIG_as_voidptr(outParam),SWIGTYPE_p
_MPI_Comm, 0 | 0 ));
        return ret;
    }

    void closeLib(PyObject*obj)
    {
        void *argp;
        int status=SWIG_ConvertPtr(obj,&argp,SWIGTYPE_p_MPI_Comm,0|0);
        closeLib(argp);
    }
%}

%extend ICoCo::Problem {
    PyObject* computeTimeStep()
    {
        bool stop;
        double dt;
        dt = (*self).computeTimeStep(stop);
        cout << "double dt" << dt << endl;
        cout << "stop " << stop << endl;
        PyObject *ret=PyTuple_New(2);
        PyTuple_SetItem(ret,0,PyFloat_FromDouble(dt));
        PyTuple_SetItem(ret,1,PyBool_FromLong(stop));
        return ret;
    }
}

```

After a compilation step, an extension module `_ICoComodule.so` is created. It can now be used in a python script.

The following paragraphs will detail the python supervisor to launch the use case described in §3. The complete python script can be found in sixth annex.

4.4.3 MPI initialization

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. MPI_COMM_WORLD is the predefined communicator that includes all of MPI processes. A group is an ordered set of processes.

A new group as a subset of global group and a new communicator are created for Trio_U code. This step allows Trio_U to manage the parallelism by itself inside its group.

The Cathare code will be launched on first processor (rank zero) and Trio_U will be launched on all the others. Therefore, Trio_U group contains all processors except the zero one.

According to the processor number, Cathare Trio_U will be launched.

Different output files are defined to avoid mixing information from the two codes.

First, we need to import ICoCoModule, created by swig tool and particularly the useful functions.

```
import ICoCoModule,os
from ICoCoModule import getFieldMean,setConstantField,openLib, closeLib
from ICoCoModule import mpi_min,mpi_or, mpi_bcast, mpi_and, my_MPI_Comm_size,
my_MPI_Comm_rank,MPI_Comm_new , my_MPI_Init
```

```
import sys
```

```
# MPI initialization
my_MPI_Init(sys.argv)
size = my_MPI_Comm_size()
rank = my_MPI_Comm_rank()
```

```
trio_ids = range(1,size)
```

```
#Create the communicators for Trio_U
comm_trio = MPI_Comm_new(size-1, trio_ids)
```

```
# Redirect the outputs
if rank == 0:
    progname="Cathare"
    libname = "./libcathare_gad.so"
    pass
else:
    progname = "Trio"
    libname = "_Trio_UModule_opt.so"
    pass
```

```
prog_out = progname + ".out"
sys.stdout = open(prog_out,"w")
prog_err = progname + ".err"
sys.stderr = open(prog_err,"w")
```

4.4.4 Instantiation of the problems

For each code, a call to function openLib described in previous paragraph is enough. If program runs on processor number 0, the problem P represents Cathare. On the other processors, the problem P represents Trio.

Documentation of the Interface for Code Coupling : ICoCo

```
# open shared libraries
P, handle =openLib(libname)
```

4.4.5 Initialization of the problems

Trio_U needs both data file and a MPI communicator. The communicator is the one created only for Trio_U code defined in §4.2.2

For Cathare code, it is quite different.

In fact, before doing a Cathare calculation process, Cathare needs a pre-processing step where data acquisition is done (description of the hydraulic circuit(s) to be simulated, the events occurring during the simulation and how calculation is managed). This pre-processing step is done and the pre-compiled data file is integrated to the dynamic library loaded at problem instantiation.

Then, the data file has already been defined through the dynamic library loaded at problem instantiation. As a consequence, Cathare doesn't need any initialization parameter.

```
# Initialize the problems
if (programe=="Trio"):
  if (size<3):
    P.setDataFile("jdd_Trio.data")
    pass
  else:
    P.setDataFile("PAR_jdd_Trio.data")
    pass
P.setMPIComm(comm_trio)
pass
P.initialize()
```

4.4.6 Initialization of coupling parameters and data to transform data

There is in this part no difference with the sequential mode.

```
# Coupling parameters
# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
#  $H = h_{ref} + C_{p\_Trio} * (T - T_{ref})$ 
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m (S=0.28274m2)
# In Trio_U, 2D pipe of section 0.6m
# We keep the same  $Q = \rho * u * S$  between Cathare and Trio_U
# =>  $u(Trio) = Q(Cathare) / S(Trio) / \rho(Trio)$ 
# !!  $u(Trio) \neq u(Cathare)$ 
S_Trio = 0.6
```

Documentation of the Interface for Code Coupling : ICoCo

```
rho_Trio = 739.206
```

```
# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1
```

4.4.7 First time loop : Cathare alone to reach a Cathare steady state

During the first 1000 seconds, Cathare will run alone to reach a Cathare steady state.

First time loop (Cathare alone)

```
epsilon=1e-10 # small time for double comparisons

if (programe=="Cathare"):
    while (1) : # Loop on timesteps
        present=P.presentTime()
        dt, stop =P.computeTimeStep()
        #stop = stop_p.value()
        if (stop or present>t_begin_trio-epsilon):
            break

        # Modify dt in order to come to exactly t_begin_trio
        if (present+dt>t_begin_trio):
            dt=t_begin_trio-present
        elif (present+2*dt>t_begin_trio):
            dt=(t_begin_trio-present)/2

        # Initialize the timestep
        P.initTimeStep(dt)

        # Perform the computation
        ok=P.solveTimeStep()

        # Either validate or abort it
        if not(ok): # The resolution failed, try with a new dt
            P.abortTimeStep()
        else: # Validate and go to the next time step
            P.validateTimeStep()
            pass # End loop on timesteps
    pass
pass
```

4.4.8 Second time loop : both codes with one-way coupling

This part contains the same than in sequential mode with moreover the data transfer between the different processors (mpi_bcast). mpi_bcast sends a variable value from one process to all other processes of the group. If the process number is zero, it means that it is a Cathare value which is sent to other processes. If the process number is one, it means that it is a Trio_U value which is sent to all other processes.

The exchanged data should be initialized by all process before being sent by one of them.

Documentation of the Interface for Code Coupling : ICoCo

The boolean values, to determine if the final time has been reached or to know is the time step calculation has correctly been performed, are calculated with the `mpi_and` and `mpi_or` functions described in §4.3.1.

Second time loop (both codes)

```
ok=True      # Is the time interval successfully solved?
stop=False
while not(stop) :                               # Loop on timesteps
    ok=False
    while not(ok) :                               # Loop on timestep size
```

Compute time step length

```
dt, stop =P.computeTimeStep()
dt=mpi_min(dt)
stop=mpi_or(stop)
```

```
# And decide if we have to stop
if (stop):
    break
```

prepare the new time step

```
P.initTimeStep(dt)
```

Get output from Cathare

```
# Pressures
```

initialization of exchanged data

```
C_pressure_inlet1 = 0.
C_pressure_outlet1 = 0.
C_pressure_inlet2 = 0.
C_pressure_outlet2 = 0.
C_massflow_inlet1 = 0.
C_massflow_outlet1 = 0.
C_massflow_inlet2 = 0.
C_massflow_outlet2 = 0.
C_enthalpy_inlet1 = 0.
C_enthalpy_outlet1 = 0.
C_enthalpy_inlet2 = 0.
C_enthalpy_outlet2 = 0.
if (progname=="Cathare") :
    C_pressure_inlet1=getFieldMean(P,"PRESSURE_PIPE11_85")
    C_pressure_outlet1=getFieldMean(P,"PRESSURE_PIPE12_1")
    C_pressure_inlet2=getFieldMean(P,"PRESSURE_PIPE21_85")
    C_pressure_outlet2=getFieldMean(P,"PRESSURE_PIPE22_1")
```

Mass flow rates

```
C_massflow_inlet1=getFieldMean(P,"LIQFLOW_PIPE11_85")
C_massflow_outlet1=getFieldMean(P,"LIQFLOW_PIPE12_1")
C_massflow_inlet2=getFieldMean(P,"LIQFLOW_PIPE21_85")
C_massflow_outlet2=getFieldMean(P,"LIQFLOW_PIPE22_1")
```

Enthalpies

```
C_enthalpy_inlet1=getFieldMean(P,"LIQH_PIPE11_85")
```

Documentation of the Interface for Code Coupling : ICoCo

```

C_enthalpy_outlet1=getFieldMean(P,"LIQH_PIPE12_1")
C_enthalpy_inlet2=getFieldMean(P,"LIQH_PIPE21_85")
C_enthalpy_outlet2=getFieldMean(P,"LIQH_PIPE22_1")
pass

```

```

C_pressure_inlet1 = mpi_bcast(C_pressure_inlet1,0)
C_pressure_outlet1 = mpi_bcast(C_pressure_outlet1,0)
C_pressure_inlet2 = mpi_bcast(C_pressure_inlet2,0)
C_pressure_outlet2 = mpi_bcast(C_pressure_outlet2,0)
C_massflow_inlet1 = mpi_bcast(C_massflow_inlet1,0)
C_massflow_outlet1 = mpi_bcast(C_massflow_outlet1,0)
C_massflow_inlet2 = mpi_bcast(C_massflow_inlet2,0)
C_massflow_outlet2 = mpi_bcast(C_massflow_outlet2,0)
C_enthalpy_inlet1 = mpi_bcast(C_enthalpy_inlet1,0)
C_enthalpy_outlet1 = mpi_bcast(C_enthalpy_outlet1,0)
C_enthalpy_inlet2 = mpi_bcast(C_enthalpy_inlet2,0)
C_enthalpy_outlet2 = mpi_bcast(C_enthalpy_outlet2,0)

```

Compute Trio_U input values

```

# calculation of input temperature for Trio_U
temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref

```

calculation of inlet and outlet velocity for Trio_U

```

u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio

```

```

# Modifications to ensure div(u)=0
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4

```

Get output from Trio_U**# initialization of exchanged data**

```

T_pressure_inlet1 = 0.
T_pressure_outlet1 = 0.
T_pressure_inlet2 = 0.
T_pressure_outlet2 = 0.
T_temperature_inlet1 = 0.
T_temperature_inlet2 = 0.
T_temperature_outlet1 = 0.
T_temperature_outlet2 = 0.
if (progname=="Trio"):
    # Pressures
    T_pressure_inlet1=getFieldMean(P,"pressure_inlet1",comm_trio)*rho_Trio
    T_pressure_inlet2=getFieldMean(P,"pressure_inlet2",comm_trio)*rho_Trio

```

Documentation of the Interface for Code Coupling : ICoCo

```
T_pressure_outlet1=getFieldMean(P,"pressure_outlet1",comm_trio)*rho_Trio
T_pressure_outlet2=getFieldMean(P,"pressure_outlet2",comm_trio)*rho_Trio

# Temperatures
T_temperature_inlet1=getFieldMean(P,"temperature_inlet1",comm_trio)
T_temperature_inlet2=getFieldMean(P,"temperature_inlet2",comm_trio)
T_temperature_outlet1=getFieldMean(P,"temperature_outlet1",comm_trio)
T_temperature_outlet2=getFieldMean(P,"temperature_outlet2",comm_trio)

pass

T_pressure_inlet1 = mpi_bcast(T_pressure_inlet1, 1)
T_pressure_outlet1 = mpi_bcast(T_pressure_outlet1, 1)
T_pressure_inlet2 = mpi_bcast(T_pressure_inlet2, 1)
T_pressure_outlet2 = mpi_bcast(T_pressure_outlet2, 1)

T_temperature_inlet1 = mpi_bcast(T_temperature_inlet1, 1)
T_temperature_inlet2 = mpi_bcast(T_temperature_inlet2, 1)
T_temperature_outlet1 = mpi_bcast(T_temperature_outlet1, 1)
T_temperature_outlet2 = mpi_bcast(T_temperature_outlet2, 1)

# Give input to Cathare
# Trio_U -> Cathare retroaction on pressure
if (programe=="Cathare"):
  if (P.presentTime(>time_retro_P) :
    dP_T1 = T_pressure_inlet1-T_pressure_outlet1
    dP_T2 = T_pressure_inlet1-T_pressure_outlet2
    dP_T3 = T_pressure_inlet1-T_pressure_inlet2

    dP_C1 = C_pressure_inlet1-C_pressure_outlet1
    dP_C2 = C_pressure_inlet1-C_pressure_outlet2
    dP_C3 = C_pressure_inlet1-C_pressure_inlet2

    tau_coupl=0.1
    k_loc=0
    if (dt>tau_coupl):
      k_loc=1
    else:
      k_loc=dt/tau_coupl

    # pressure retroaction on outlets
    dP1 += (dP_C1-dP_T1)*k_loc
    dP2 += (dP_C2-dP_T2)*k_loc
    dP3 -= (dP_C3-dP_T3)*k_loc

    setConstantField(P,"DPLEXT_PIPE12_1",dP1)
    setConstantField(P,"DPLEXT_PIPE22_1",dP2)
    setConstantField(P,"DPLEXT_PIPE21_86",dP3)
  pass
```


*Documentation of the Interface for Code Coupling : ICoCo***# Trio_U -> Cathare retroaction on temperature**

```
if (P.presentTime(>time_retro_T) :
  T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
  T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
  T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
  T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)

  # mesh numbers are growing from "left" to "right"
  # overlapping domain on the "right"
  # Loop 1
  setConstantField(P,"OVHLEXT_PIPE11_85",-10)
  setConstantField(P,"OVPLEXT_PIPE11_85",-10)
  setConstantField(P,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1)
  # Loop 2
  setConstantField(P,"OVHLEXT_PIPE21_85",-10)
  setConstantField(P,"OVPLEXT_PIPE21_85",-10)
  setConstantField(P,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2)

  # overlapping domain on the "left"
  # Loop 1
  setConstantField(P,"OVHLEXT_PIPE12_1",10)
  setConstantField(P,"OVPLEXT_PIPE12_2",10)
  setConstantField(P,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1)
  # Loop 2
  setConstantField(P,"OVHLEXT_PIPE22_1",10)
  setConstantField(P,"OVPLEXT_PIPE22_2",10)
  setConstantField(P,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2)
  pass
pass

# Give input to Trio_U
# Velocities
if (programe=="Trio"):
  setConstantField(P,"u_inlet1",u_inlet1,1)
  setConstantField(P,"u_inlet2",u_inlet2,1)
  setConstantField(P,"u_outlet1",u_outlet1,1)
  setConstantField(P,"u_outlet2",u_outlet2,1)

  # Temperatures
  setConstantField(P,"temperature_inlet1",temperature_inlet1)
  setConstantField(P,"temperature_inlet2",temperature_inlet2)
  setConstantField(P,"temperature_outlet1",temperature_outlet1)
  setConstantField(P,"temperature_outlet2",temperature_outlet2)
  pass

# Solve next time step
ok =P.solveTimeStep()
ok = mpi_and(ok)
# Two ways : The resolution failed, try with a new dt or validate and go
# to next time step
if not(ok) :
  P.abortTimeStep()
else:
```

Documentation of the Interface for Code Coupling : ICoCo

```
P.validateTimeStep()
```

```
    pass                                # End loop on timestep size  
pass                                  # End loop on timesteps
```

4.4.9 End of the coupling calculation

There is no difference with the sequential mode.

```
P.terminate()
```

```
closeLib(handle)
```

5 Use of the generic interface ICoCo with SALOME platform

The first thing to do is to integrate a new module to the SALOME platform: the ICoCo component. The component should provide all services of the ICoCo interface and services which correspond to the useful functions seen in user developer supervisor.

These functions concern:

- opening the code dynamic library and instantiation of a ICoCo problem: this is done in *OpenLib* function
- calculation of the mean value of a one component field: the *getFieldMean* function.
- affectation of all cells of a one component field with the same value: the function *setConstantField*.

The following paragraphs will describe:

- first, how to construct the C++ Salome ICoCo component,
- then, how to use Salome ICoCo component in a python script,
- finally, how to use Salome ICoCo component in Salome YACS module which allows build, edit and execute calculation schemes.

These last two use of Salome component will be illustrated with the use case described in §3.

5.1 Construction of the Salome ICoCo component

The easiest way to build a new Salome component is to use hxx2salome, a tool for automatic SALOME Component generation. This tool starts from the interface of a C++ component (an .hxx file), parse the public API, and use the type information to generate the SALOME component (the IDL interface, and its implementation).

A C++ component is a “high level” unit of reuse, based upon some source code libraries. It takes the form of a C++ class. Its interface is the public API of this class, and is declared in an include file. It is designed to collaborate with other components. Therefore its API emphasizes the logical chains of computation a user can perform, and the data that may be exchanged with external world conform to standards.

The hxx2salome tool has mandatory arguments: the path where the C++ component was installed, the name of the interface header and the name of the dynamic library, and last the location where to generate and compile the Salome component. The problem with the generic interface ICoCo is that the shared library containing the code must provide a way to retrieve the object of type Problem and thus is loaded at coupling execution, with one dynamic library by ICoCo problem. It is not compatible with Salome component generated via hxx2salome tool.

Therefore, one decides to build a Salome specific C++ component called ICoCoComponent. The Salome component will provide a function to specify the dynamic library to load and also two private attributes `_theProblem` which will store the C++ ICoCo Problem and `_theHandle` which will store the opaque handle for the dynamic library.

The header file contains all the services of an ICoCo problem, the four useful functions `openLib`, `closeLib`, `getFieldMean` and `setConstantField` and the private attributes to store the C++ ICoCo Problem and the handle to the dynamic library.

The source file of the ICoCoComponent class contains the source for the four useful functions. Note that the `openLib` function initializes the private attributes `_theProblem` and `_theHandle`. All the other functions of the class just make a call of same function of the ICoCo Problem through the private attribute `_theProblem`. The annex number seven contains the header and the source file of the Salome ICoCoComponent.

The header file should be in a directory called include.

Documentation of the Interface for Code Coupling : ICoCo

The compilation of the C++ component with ICoCo problem sources permits to construct a dynamic library called libICoCoComponent.so. The dynamic library has to be in a directory called lib.

After this step, the C++ component is done.

To use it in Salome platform, one just has to construct the Salome component with hxx2salome tool.

The command to run the script is (supposing HXX2SALOME_ROOT_DIR is in your PATH) :

```
hxx2salome -c ICoCoComponent ICoCoComponent.hxx libICoCoComponent.so  
ICoCoComponentComp
```

where the arguments mean:

- -c : to compile the component after code generation,
- ICoCoComponent: the installation directory (absolute path) of the c++ standalone component,
- ICoCoComponent.hxx: the header name of the component,
- libICoCoComponent.so: the name of the shared library,
- ICoCoComponentComp : the directory where the generated Salome component will be installed

After the execution of the hxx2salome tool, one just has to install the Salome component and to source the environment file that has been created by the tool.

The Salome ICoCoComponent can now be used in Salome platform.

5.2 Use of the Salome ICoCo component in a python script

The use case will be written in a python script called 'test.py' for example and this script will be launched with Salome thanks to the command:

```
runSalome -modules=ICoCoComponent -execute=test.py -t
```

The python script is quite the same than when ICoCo is directly used in a python supervisor. The differences concern the initialization of the problems, the call of the useful functions (no problem in input parameter) and the boolean use.

The complete python script can be found in annex number eight.

5.2.1 Instantiation of the problems

The Python module salome.py provides a functionality to access main SALOME features from the Python console (either embedded in GUI desktop or external one).

To use salome.py module, import it into the Python interpreter and initialize it by calling salome_init() function.

The Life Cycle CORBA class instance is used to get access to CORBA engine part of some SALOME module, available in the current SALOME session.

At each code (Cathare and Trio_U) corresponds a reference to the ICoCoComponent module engine. Each ICoCoComponent module engine is loaded in different the SALOME containers.

```
import salome  
import ICoCoComponent_ORB  
salome.salome_init()
```

```
T = salome.lcc.FindOrLoadComponent( "FactoryServer" , "ICoCoComponent" )  
C = salome.lcc.FindOrLoadComponent( "FactoryServer2" , "ICoCoComponent" )  
# open shared libraries  
T.openLib( "./_Trio_UModule_opt.so" )  
C.openLib( "./libcathare_gad.so" )
```

5.2.2 Initialization of the problems

This part is exactly the same than in §4.3.3 with a standalone python supervisor.

```
# Initialize the problems
T.setDataFile("jdd_Trio.data")
T.initialize()
C.initialize()
```

5.2.3 Initialization of coupling parameters and data to transform data

This paragraph contains exactly the same as for a standalone python supervisor.

```
# Coupling parameters
# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
#  $H = h_{ref} + C_p \cdot (T - T_{ref})$ 
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m ( $S=0.28274m^2$ )
# In Trio_U, 2D pipe of section 0.6m
# We keep the same  $Q=\rho \cdot u \cdot S$  between Cathare and Trio_U
# =>  $u(Trio) = Q(Cathare) / S(Trio) / \rho(Trio)$ 
# !!  $u(Trio) \neq u(Cathare)$ 
S_Trio = 0.6
rho_Trio = 739.206

# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1
```

5.2.4 First time loop : Cathare alone to reach a Cathare steady state

This paragraph contains exactly the same as for a standalone python supervisor.

```
# First time loop (Cathare alone)
epsilon=1e-10 # small time for double comparisons

while (1) : # Loop on timesteps
    present=C.presentTime()
    dt,stop =C.computeTimeStep()
```

Documentation of the Interface for Code Coupling : ICoCo

```
if (stop or present>t_begin_trio-epsilon):
    break

# Modify dt in order to come to exactly t_begin_trio
if (present+dt>t_begin_trio):
    dt=t_begin_trio-present
elif (present+2*dt>t_begin_trio):
    dt=(t_begin_trio-present)/2

# Initialize the timestep
C.initTimeStep(dt)

# Perform the computation
ok=C.solveTimeStep()

# Either validate or abort it
if not(ok): # The resolution failed, try with a new dt
    C.abortTimeStep()
else:      # Validate and go to the next time step
    C.validateTimeStep()
pass      # End loop on timesteps
```

5.2.5 Second time loop : both codes with one-way coupling

In this part, the differences with the standalone python supervisor the call to getFieldMean and setConstantField functions.

```
# Second time loop (both codes)
ok=True      # Is the time interval successfully solved?
stop=False

while not(stop) :                                # Loop on timesteps
    ok=False
    while not(ok) :                               # Loop on timestep size

        # Compute time step length
        dt_C, stop_C=C.computeTimeStep()
        dt_T, stop_T=T.computeTimeStep()
        dt=min(dt_C,dt_T)

        # And decide if we have to stop
        stop = stop_T or stop_C
        if (stop):
            break

        # prepare the new time step
        C.initTimeStep(dt)
        T.initTimeStep(dt)

        # Get output from Cathare
        # Pressures
```

Documentation of the Interface for Code Coupling : ICoCo

```

C_pressure_inlet1=C.getFieldMean("PRESSURE_PIPE11_85")
C_pressure_outlet1=C.getFieldMean("PRESSURE_PIPE12_1")
C_pressure_inlet2=C.getFieldMean("PRESSURE_PIPE21_85")
C_pressure_outlet2=C.getFieldMean("PRESSURE_PIPE22_1")

# Mass flow rates
C_massflow_inlet1=C.getFieldMean("LIQFLOW_PIPE11_85")
C_massflow_outlet1=C.getFieldMean("LIQFLOW_PIPE12_1")
C_massflow_inlet2=C.getFieldMean("LIQFLOW_PIPE21_85")
C_massflow_outlet2=C.getFieldMean("LIQFLOW_PIPE22_1")

# Enthalpies
C_enthalpy_inlet1=C.getFieldMean("LIQH_PIPE11_85")
C_enthalpy_outlet1=C.getFieldMean("LIQH_PIPE12_1")
C_enthalpy_inlet2=C.getFieldMean("LIQH_PIPE21_85")
C_enthalpy_outlet2=C.getFieldMean("LIQH_PIPE22_1")

# Compute Trio_U input values
#calculation of input temperature for Trio_U
temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref

# calculation of inlet and outlet velocity for Trio_U
u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio

# Modifications to ensure div(u)=0
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4

# Get output from Trio_U

# Pressures
T_pressure_inlet1=T.getFieldMean("pressure_inlet1")*rho_Trio
T_pressure_inlet2=T.getFieldMean("pressure_inlet2")*rho_Trio
T_pressure_outlet1=T.getFieldMean("pressure_outlet1")*rho_Trio
T_pressure_outlet2=T.getFieldMean("pressure_outlet2")*rho_Trio

# Temperatures
T_temperature_inlet1=T.getFieldMean("temperature_inlet1")
T_temperature_inlet2=T.getFieldMean("temperature_inlet2")
T_temperature_outlet1=T.getFieldMean("temperature_outlet1")
T_temperature_outlet2=T.getFieldMean("temperature_outlet2")

# Give input to Cathare

```

Documentation of the Interface for Code Coupling : ICoCo

```
# Trio_U -> Cathare retroaction on pressure
if (C.presentTime(>time_retro_P) :
  dP_T1 = T_pressure_inlet1-T_pressure_outlet1
  dP_T2 = T_pressure_inlet1-T_pressure_outlet2
  dP_T3 = T_pressure_inlet1-T_pressure_inlet2

  dP_C1 = C_pressure_inlet1-C_pressure_outlet1
  dP_C2 = C_pressure_inlet1-C_pressure_outlet2
  dP_C3 = C_pressure_inlet1-C_pressure_inlet2

  tau_coupl=0.1
  k_loc=0
  if (dt>tau_coupl):
    k_loc=1
  else:
    k_loc=dt/tau_coupl

# pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc
dP2 += (dP_C2-dP_T2)*k_loc
dP3 -= (dP_C3-dP_T3)*k_loc
C.setConstantField("DPLEXT_PIPE12_1",dP1,0)
C.setConstantField("DPLEXT_PIPE22_1",dP2,0)
C.setConstantField("DPLEXT_PIPE21_86",dP3,0)
pass

# Trio_U -> Cathare retroaction on temperature
if (C.presentTime(>time_retro_T) :
  T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
  T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
  T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
  T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)

# mesh numbers are growing from "left" to "right"
# overlapping domain on the "right"
# Loop 1
C.setConstantField("OVHLEXT_PIPE11_85",-10,0)
C.setConstantField("OVPLEXT_PIPE11_85",-10,0)
C.setConstantField("ENTHEXT_PIPE11_85",T_enthalpie_inlet1,0)
# Loop 2
C.setConstantField("OVHLEXT_PIPE21_85",-10,0)
C.setConstantField("OVPLEXT_PIPE21_85",-10,0)
C.setConstantField("ENTHEXT_PIPE21_85",T_enthalpie_inlet2,0)

# overlapping domain on the "left"
# Loop 1
C.setConstantField("OVHLEXT_PIPE12_1",10,0)
C.setConstantField("OVPLEXT_PIPE12_2",10,0)
C.setConstantField("ENTHEXT_PIPE12_1",T_enthalpie_outlet1,0)
# Loop 2
C.setConstantField("OVHLEXT_PIPE22_1",10,0)
C.setConstantField("OVPLEXT_PIPE22_2",10,0)
C.setConstantField("ENTHEXT_PIPE22_1",T_enthalpie_outlet2,0)
```


Documentation of the Interface for Code Coupling : ICoCo

```
pass
```

```
# Give input to Trio_U
```

```
# Velocities
```

```
print 'initialisation variables Trio'
```

```
T.setConstantField("u_inlet1",u_inlet1,1)
```

```
T.setConstantField("u_inlet2",u_inlet2,1)
```

```
T.setConstantField("u_outlet1",u_outlet1,1)
```

```
T.setConstantField("u_outlet2",u_outlet2,1)
```

```
# Temperatures
```

```
T.setConstantField("temperature_inlet1",temperature_inlet1,0)
```

```
T.setConstantField("temperature_inlet2",temperature_inlet2,0)
```

```
T.setConstantField("temperature_outlet1",temperature_outlet1,0)
```

```
T.setConstantField("temperature_outlet2",temperature_outlet2,0)
```

```
# Solve next time step
```

```
ok_T=T.solveTimeStep()
```

```
ok_C=C.solveTimeStep()
```

```
ok = ok_T and ok_C
```

```
# Two ways : The resolution failed, try with a new dt or validate and go
```

```
# to next time step
```

```
if not(ok) :
```

```
    T.abortTimeStep()
```

```
    C.abortTimeStep()
```

```
    print "Abort at time " + C.presentTime()
```

```
else:
```

```
    T.validateTimeStep()
```

```
    C.validateTimeStep()
```

```
pass
```

```
# End loop on timestep size
```

```
pass
```

```
# End loop on timesteps
```

5.2.6 End of the coupling calculation

```
T.terminate()
```

```
C.terminate()
```

```
T.closeLib()
```

```
C.closeLib()
```

5.3 Use of the Salome ICoCo component in Salome YACS module

YACS is a tool for managing multidisciplinary simulations through calculation schemes.

The YACS module can be used to build and execute calculation schemes. A calculation scheme is a more or less complex assembly of calculation components (SALOME components or calculation codes or python scripts). Therefore, a calculation scheme provides a means of defining a chain or coupling of calculation codes .

A calculation scheme can be built either using a graphic, or by editing an XML file, or by using an application programming interface (API) in Python. In this phase, chaining of components is defined with the associated dataflows.

The calculation scheme can be executed from the graphic tool, but also in console mode, or by using the Python interface.

Executing a calculation scheme includes:

- running and distributing components
- managing data distribution
- monitoring execution
- stopping / suspending / restarting the execution

A calculation scheme is constructed based on the calculation node concept. A calculation node represents an elementary calculation that can be the local execution of a Python script or the remote execution of a SALOME component service.

The calculation scheme is a more or less complex assembly of calculation nodes.

This assembly is made by connecting input and output ports of these calculation nodes.

Data are exchanged between nodes through ports. They are typed.

Composite nodes: Block, Loop, Switch are used to modularise a calculation scheme and define iterative processes, parametric calculations or branches.

Finally, containers can be used to define where SALOME components will be executed (on a network or in a cluster).

The calculation scheme will be saved in a xml file called 'doubleloop.xml' for example and this calculation scheme will be executed from the graphic tool or in console mode with Salome thanks to the command:

```
runSalome --modules=YACS, ICoCoComponent --execute=Test_Yacs.py -t
```

with Test_Yacs.py containing :

```
import loader
import pilot
import SALOMERuntime
import threading
```

```
SALOMERuntime.RuntimeSALOME_setRuntime(True)
runtime=pilot.getRuntime()
xmlLoader = loader.YACSLoader()
mainProc2L = xmlLoader.load("doubleloop.xml")
exe=pilot.ExecutorSwig()
exe.RunPy(mainProc2L)
```

The built of the calculation scheme of this use case is quite long and complicated.

The complete calculation scheme is shown in Figure 5 and will be detailed in further paragraphs.

Documentation of the Interface for Code Coupling : ICoCo

The blue lines represent data connections between different ports. The purple lines represent links which are used to define an order in which nodes will be executed.

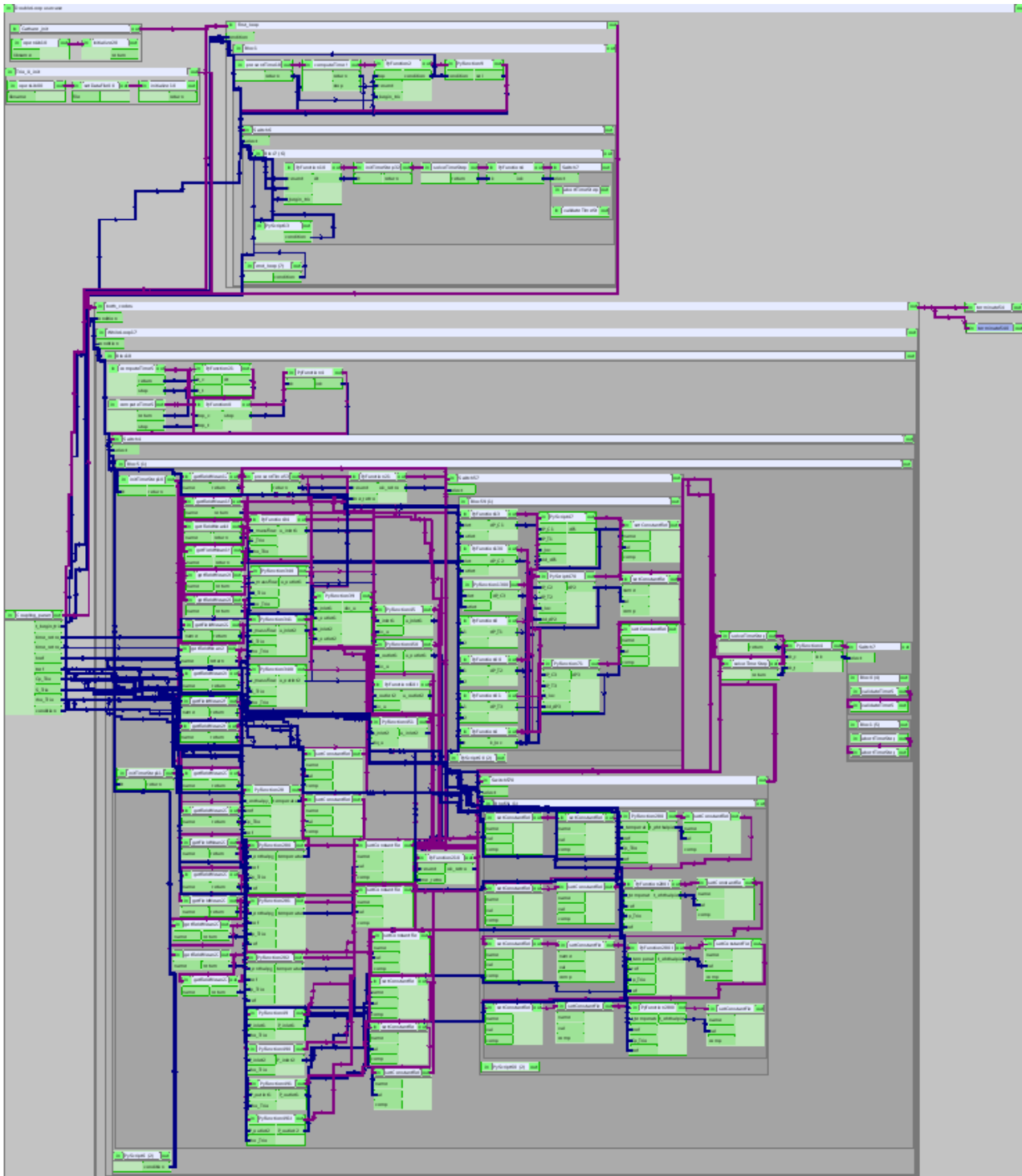


FIGURE 5 : YACS COMPLETE CALCULATION SCHEME

5.3.1 Instantiation and initialization of the problems

Two bloc nodes are defined, one by Salome ICoCo Component. Each Salome ICoCo component is executed on a separate container. For each defined node, mind to the Salome component instance

name which is always a new one. Each time, you have to choose one of the two instance created with openLib node.

The Cathare instantiation and initialization bloc contains one call to the openLib node and one call to the initialize node.

The Trio_U instantiation and initialization bloc contains one call to the openLib node, one call to setDataFile node and one call to the initialize node.

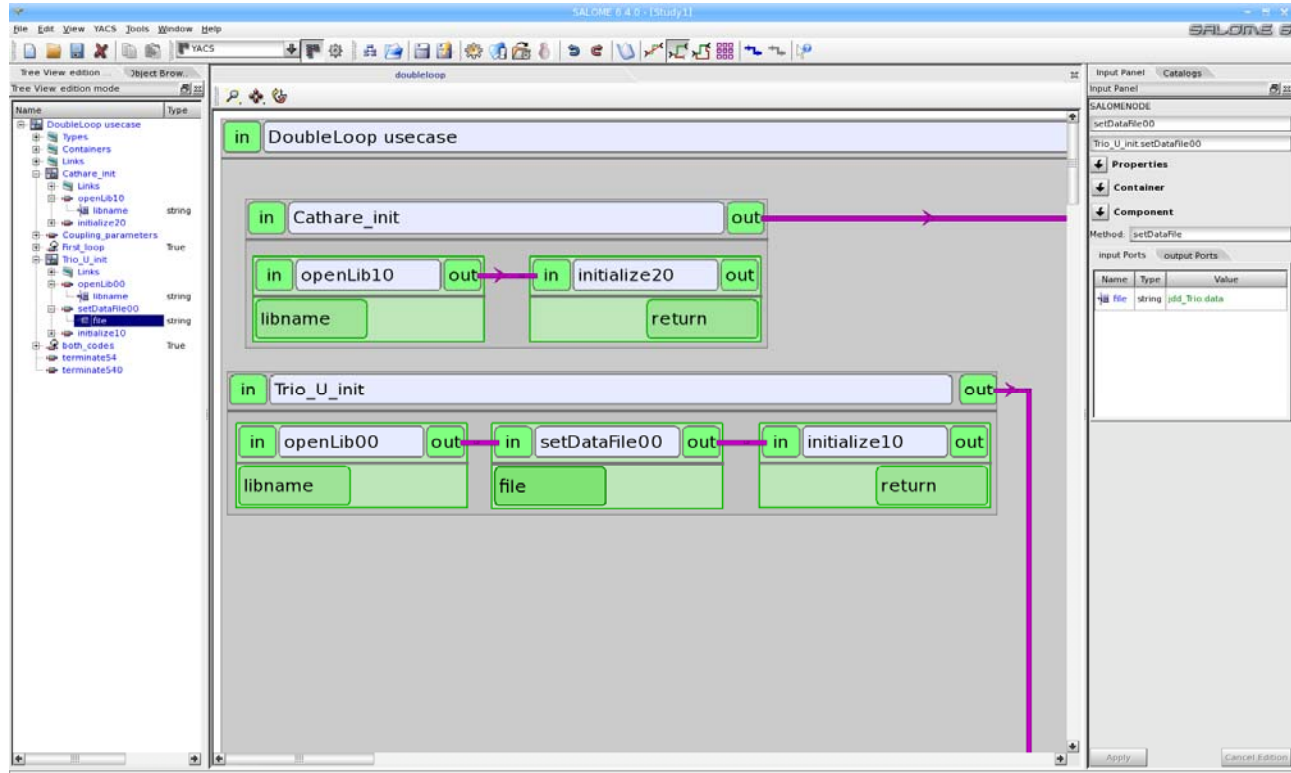


FIGURE 6 : YACS INSTANTIATION AND INITIALIZATION OF THE PROBLEMS

5.3.2 Initialization of coupling parameters and data to transform data

The definition of the coupling parameters and internal data is made with a PyScript node. All the necessary data are defined as output ports and can then be used by all the others scheme nodes.

Documentation of the Interface for Code Coupling : ICoCo

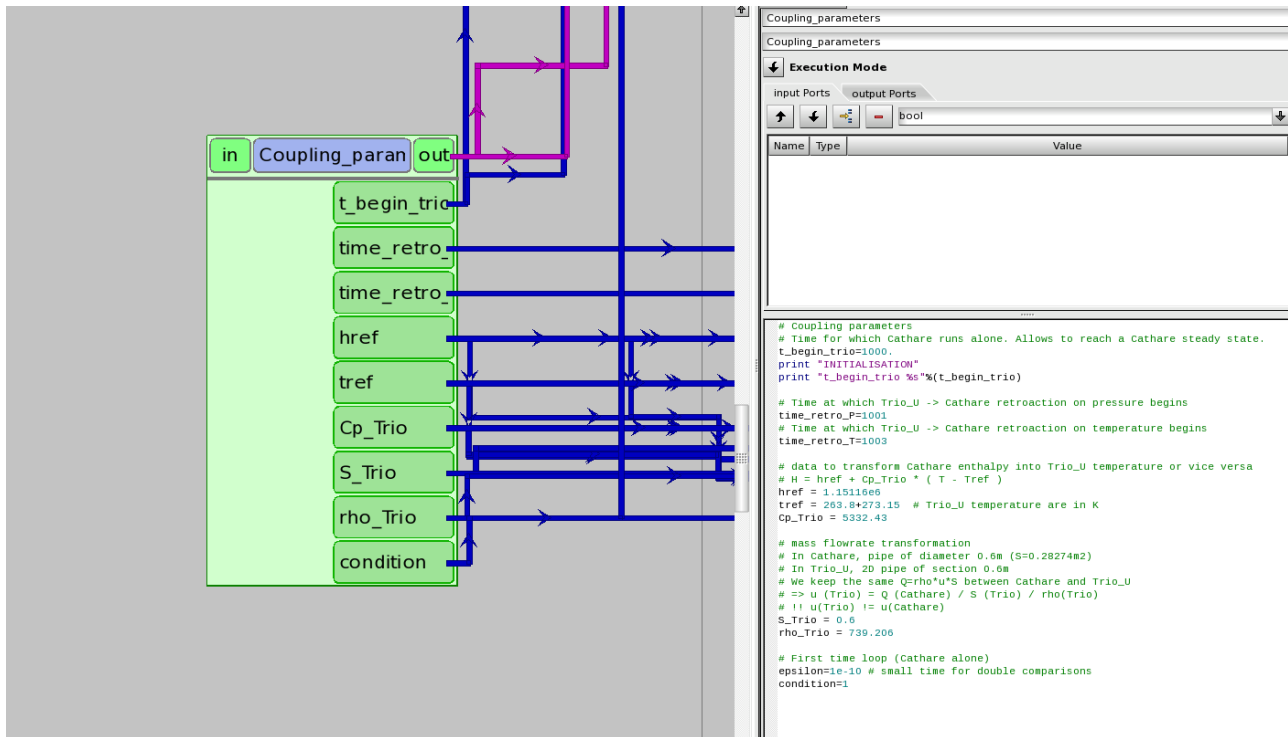


FIGURE 7 : YACS INITIALIZATION OF DATA

5.3.3 First time loop : Cathare alone to reach a Cathare steady state

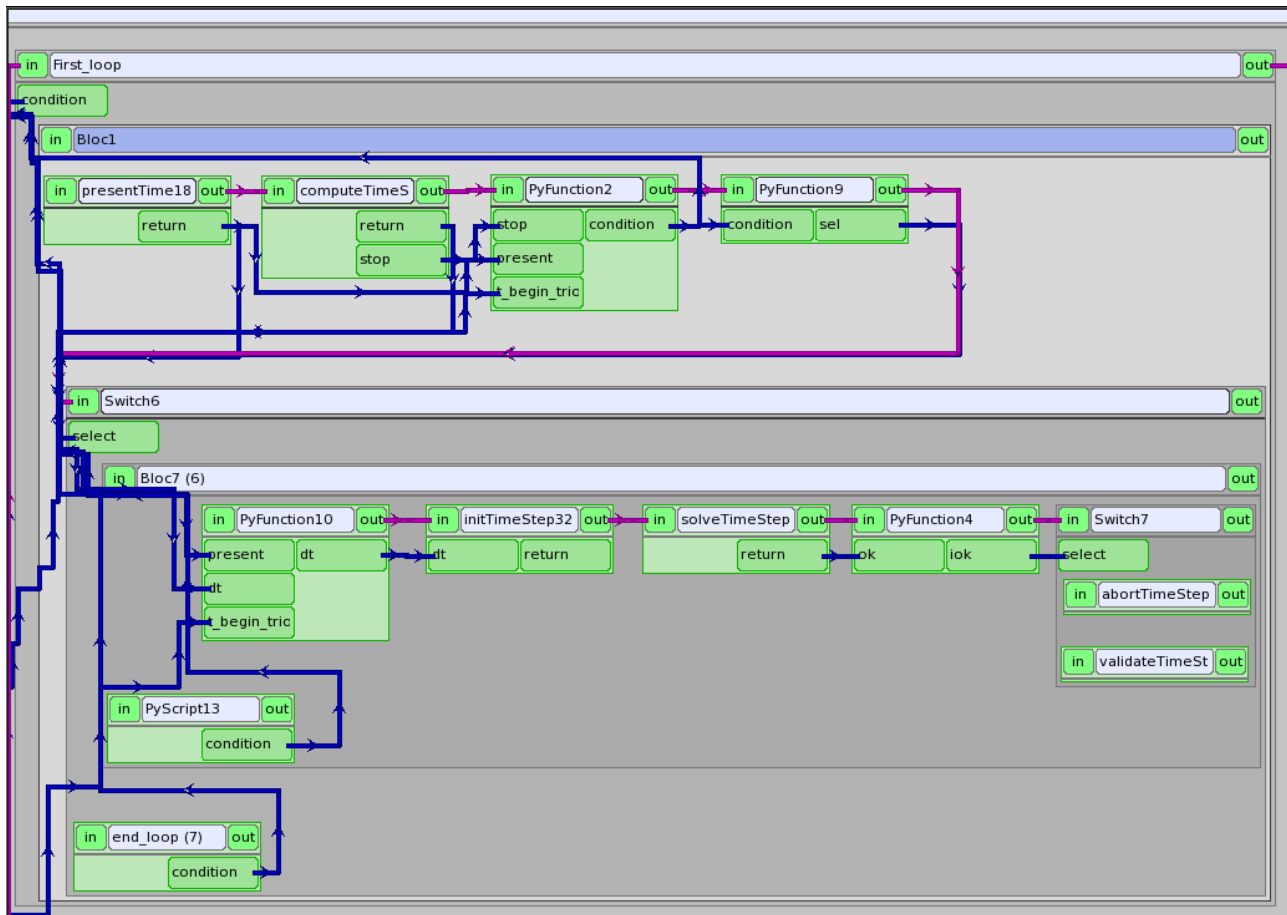


FIGURE 8 : YACS FIRST TIME LOOP

5.3.4 Second time loop : both codes with one-way coupling

This part is really complicated. There is one node for each getFieldMean or setConstantField call, one PyFunction node for each data transformation (enthalpy <-> temperature) for example and two blocs to treat if the different retroaction times have been reached or not.

Documentation of the Interface for Code Coupling : ICoCo

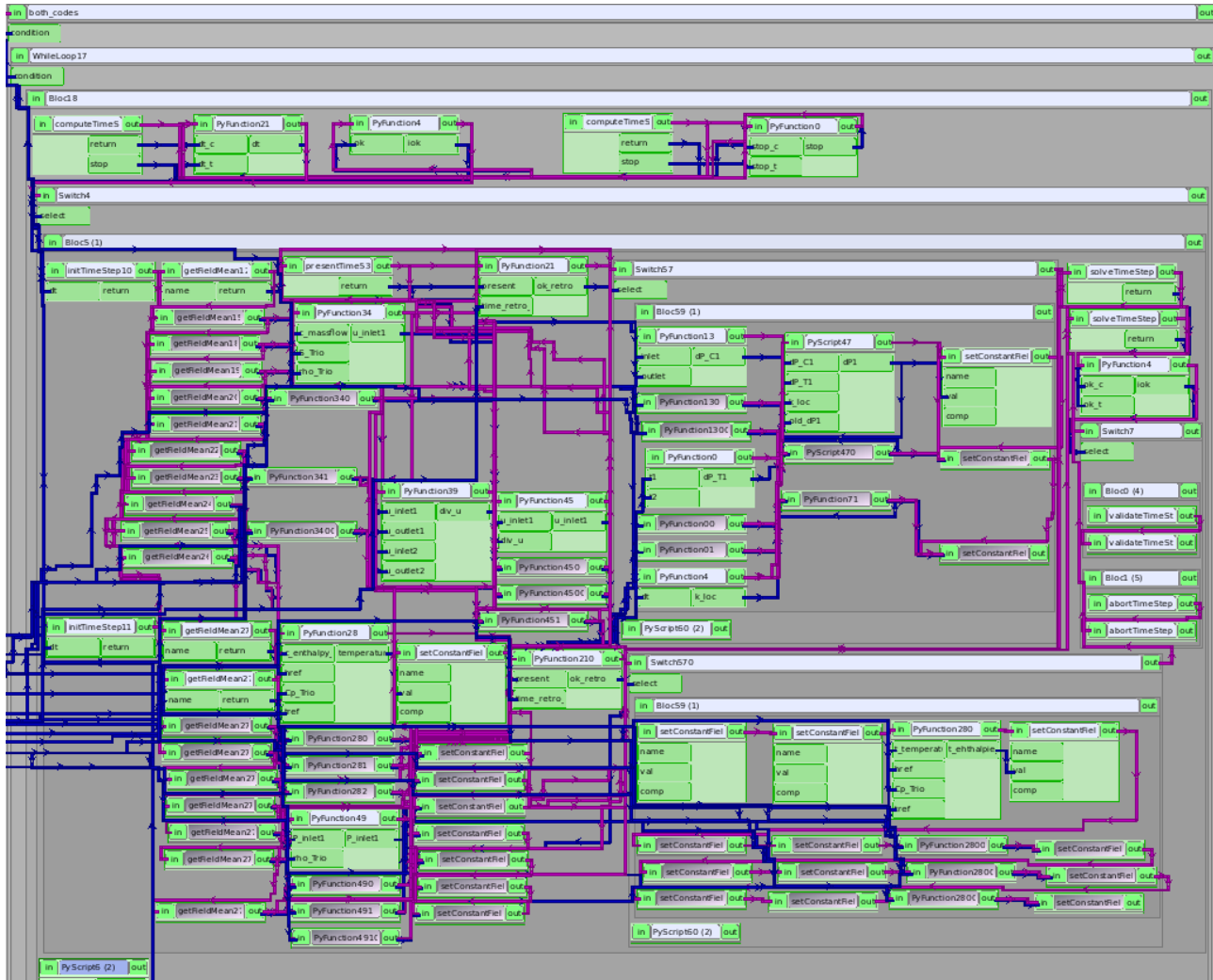
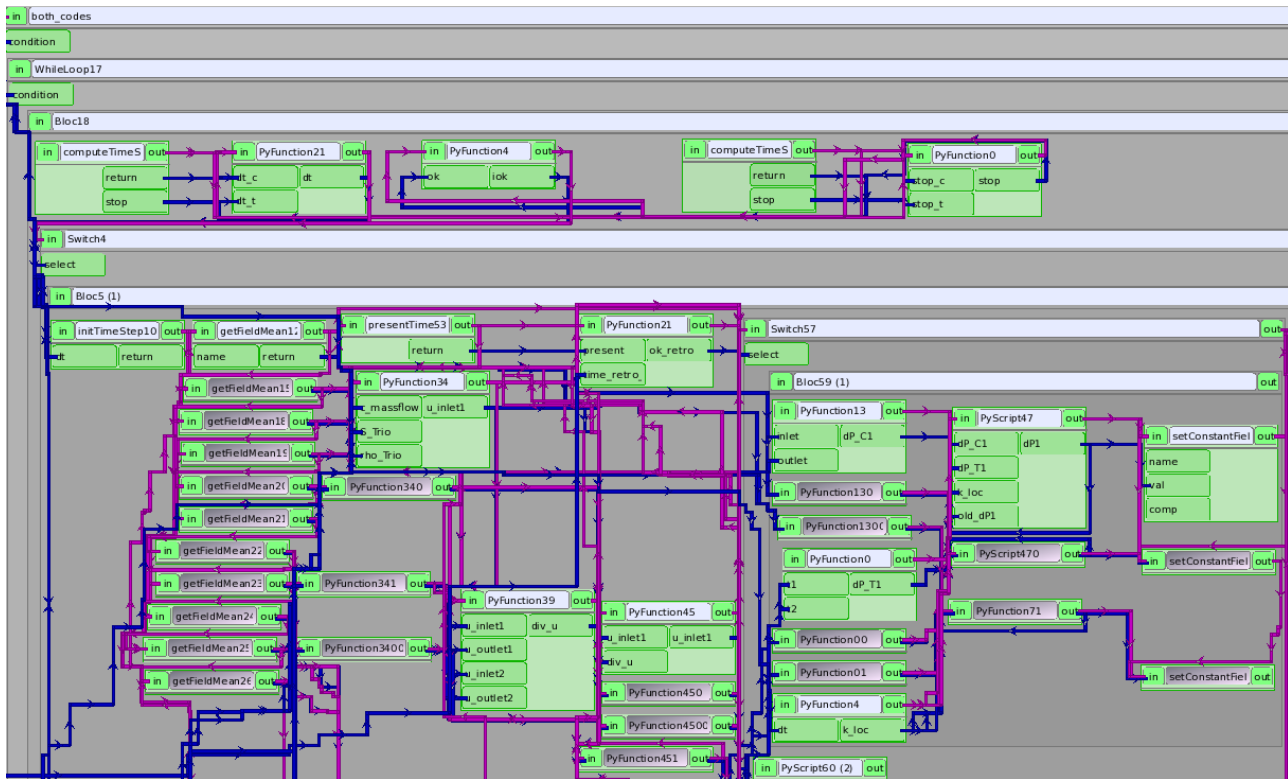


FIGURE 9 : YACS SECOND TIME LOOP

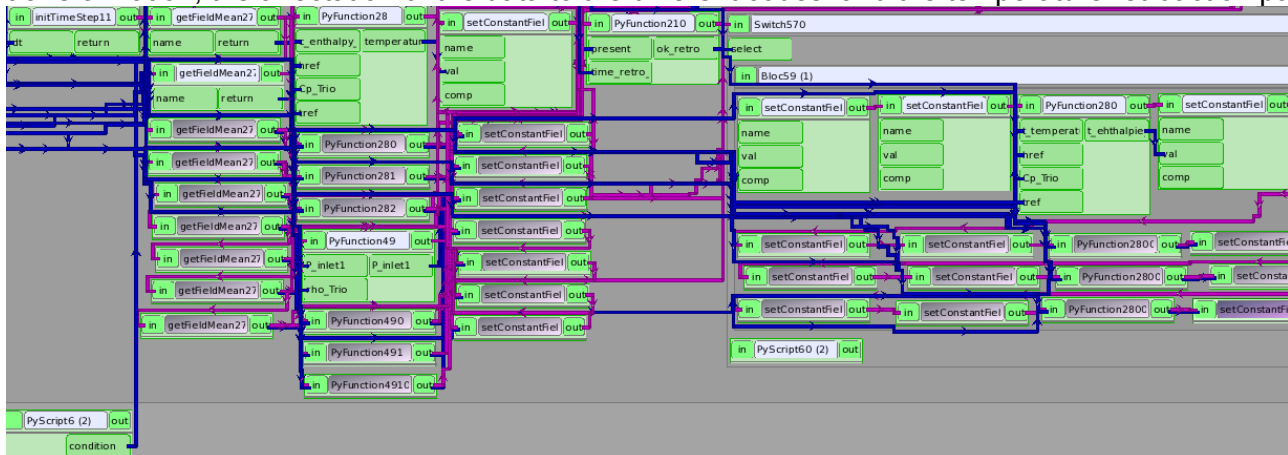
The 3 following figures are made by splitting the previous bloc in 3 parts.

The first one contains Cathare and Trio_U call to computeTimeStep, Cathare time step initialization, the recuperation of the Cathare outputs, the data transformation and the pressure retroaction part.

Documentation of the Interface for Code Coupling : ICoCo

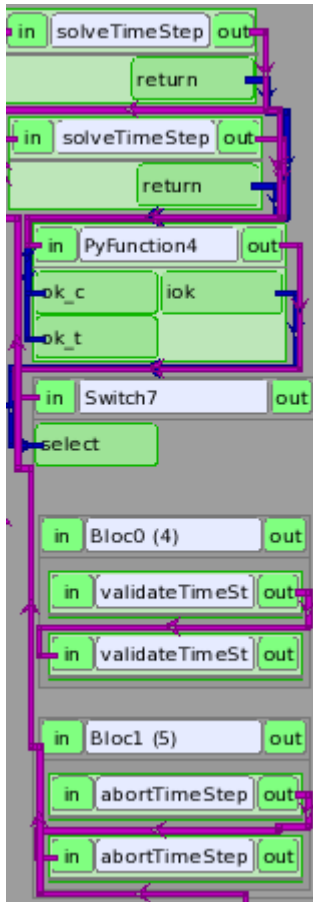


The second one contains Trio_U time step initialization, the recuperation of the Trio_U outputs, the data transformation, the affectation of the data to the different codes and the temperature retroaction part.



The third part contains call to solveTimeStep, validateTimeStep or abortTimeStep for each code.

Documentation of the Interface for Code Coupling : ICoCo



5.3.5 End of the coupling calculation

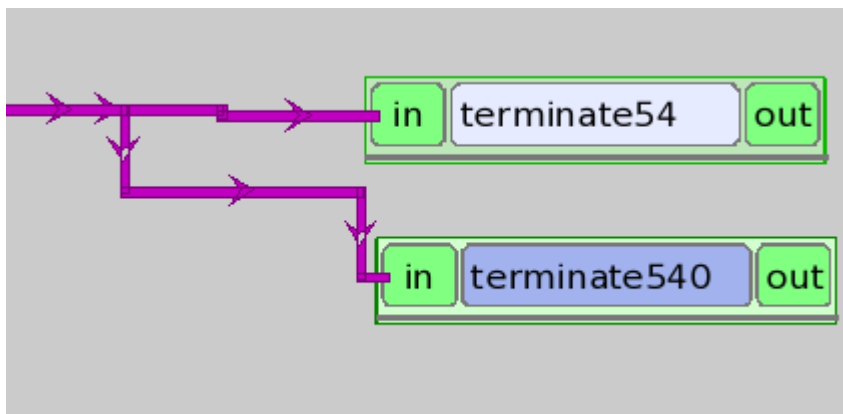


FIGURE 10 : YACS END OF COUPLING CALCULATION

Annexes

5.4 Annex 1: sources of ICoCo interface

5.4.1 Header file: Problem.h

```
#ifndef _Problem_included_
#define _Problem_included_

#include <vector>
#include <string>

namespace ParaMEDMEM {
    class MEDCouplingFieldDouble;
}

namespace ICoCo {
    class TrioField;
    class Problem {
    public :

        // interface Problem
        Problem();
        virtual ~Problem();
        virtual void setDataFile(const std::string& datafile);
        virtual void setMPIComm(void* mpicomm);
        virtual bool initialize();
        virtual void terminate();

        // interface UnsteadyProblem
        virtual double presentTime() const;
        virtual double computeTimeStep(bool& stop) const;
        virtual bool initTimeStep(double dt);
        virtual bool solveTimeStep();
        virtual void validateTimeStep();
        virtual bool isStationary() const;
        virtual void abortTimeStep();

        // interface IterativeUnsteadyProblem
        virtual bool iterateTimeStep(bool& converged);

        // interface Restorable
        virtual void save(int label, const std::string& method) const;
        virtual void restore(int label, const std::string& method);
        virtual void forget(int label, const std::string& method) const;

        // interface FieldIO
        virtual std::vector<std::string> getInputFieldsNames() const;
        virtual void getInputFieldTemplate(const std::string& name, TrioField&
        afield) const;
    };
};
```

Documentation of the Interface for Code Coupling : ICoCo

```
virtual void setInputField(const std::string& name, const TrioField&
afield);
virtual std::vector<std::string> getOutputFieldsNames() const;
virtual void getOutputField(const std::string& name, TrioField& afield)
const;

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;
};
}
extern "C" ICoCo::Problem* getProblem();
#endif
```

5.4.2 Source file: Problem.cpp

```
#include <Problem.h>
#include <Exceptions.h>

using namespace ICoCo;
using std::string;
using std::vector;
using ParamEDMEM::MEDCouplingFieldDouble;

Problem::Problem() { }
Problem::~Problem() { }

// Specify the data file used for this Problem.
// Precondition: Problem is not initialized.
// datafile: name of a datafile for the code.
void Problem::setDataFile(const string& datafile) {
    throw NotImplemented("type_of_Problem_not_set","setDataFile");
}

// Specify the MPI communicator this Problem should use internally
// For sequential calculation, mpicomm is 0.
// Precondition: Problem is not initialized.
// mpicom: the MPI communicator to be used internally.
void Problem::setMPIComm(void* mpicomm)
{
    if (mpicomm!=0)
        throw NotImplemented("type_of_Problem_not_set","setMPIComm with
comm<>0");
}

// This method must be called before any other one except
// setDataFile and setMPIComm.
// Precondition: Problem is not initialized.
// setDataFile & setMPIComm should have been called.
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Return value: true=OK, false=error
bool Problem::initialize() {
    throw NotImplemented("type_of_Problem_not_set","initialize");
    return false;
}

// This method is called once at the end, after any other one.
// It frees the memory and saves anything that needs to be saved.
// Precondition: initialize, but not yet terminate.
// Postcondition: the object is ready to be destroyed or initialized again.
void Problem::terminate() {
    throw NotImplemented("type_of_Problem_not_set","terminate");
}

// Returns the present time.
// This value may change only at the call of validateTimeStep.
// Precondition: initialize, not yet terminate
double Problem::presentTime() const {
    throw NotImplemented("type_of_Problem_not_set","presentTime");
    return 0;
}

// Compute the value the Problem would like for the next time step.
// This value must not necessarily be used for the next call to initTimeStep,
// but it is a hint.
// Precondition: initialize, not yet terminate
// stop: return value indicating if the Problem wants to stop
// return value: the desired time step
double Problem::computeTimeStep(bool& stop) const {
    throw NotImplemented("type_of_Problem_not_set","computeTimeStep");
    return 0;
}

// This method allocates and initializes the future time step.
// The value of the interval is imposed through the parameter dt.
// Precondition: initialize, not yet terminate, timestep not yet initialized,
dt>0
// dt: the time interval to allocate
// return value: true=OK, false=error, not able to tackle this dt
// Postcondition: Enables the call to several methods for the next time step
bool Problem::initTimeStep(double dt) {
    throw NotImplemented("type_of_Problem_not_set","initTimeStep");
    return false;
}

// Calculates the unknown fields for the next time step.
// Precondition: initTimeStep
// Return value: true=OK, false=unable to find the solution.
// Postcondition: The unknowns are updated over the next time step.
bool Problem::solveTimeStep() {
    throw NotImplemented("type_of_Problem_not_set","solveTimeStep");
    return false;
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Validates the calculated unknown by moving the present time
// at the end of the time step.
// This method can free past values of the fields (point of no return)
// Precondition: initTimeStep
// Postcondition: The present time has moved forward.
void Problem::validateTimeStep() {
    throw NotImplemented("type_of_Problem_not_set","terminate");
}

// Tells if the Problem unknowns have changed during the last time step.
// Precondition: validateTimeStep, not yet initTimeStep
// Return value: true=stationary, false=not stationary
bool Problem::isStationary() const {
    throw NotImplemented("type_of_Problem_not_set","isStationary");
    return false;
}

// Aborts the resolution of the current time step.
// Precondition: initTimeStep
// Postcondition: Can call initTimeStep again with a new dt.
void Problem::abortTimeStep() {
    throw NotImplemented("type_of_Problem_not_set","abortTimeStep");
}

// In the case solveTimeStep uses an iterative process,
// this method executes a single iteration.
// It is thus possible to modify the given fields between iterations.
// converged is set to true if the process has converged, ie if the
// unknown fields are solution to the problem on the next time step.
// Otherwise converged is set to false.
// The return value indicates if the convergence process behaves normally.
// If false, the Problem wishes to abort the time step resolution.
// Precondition: initTimeStep
// converged: return value, true if the process has converged, false if not
// yet
// return value: true=OK, false=unable to converge
// Postcondition: The unknowns are updated over the next time step.
bool Problem::iterateTimeStep(bool& converged) {
    throw NotImplemented("type_of_Problem_not_set","iterateTimeStep");
    return false;
}

// Save the state of the Problem.
// Precondition: initialize, not yet terminate
// label : an int identifying the saved state for a future restore
// method : the saving method
// Postcondition: The state is saved and can be restored at any time
void Problem::save(int label, const string& method) const {
    throw NotImplemented("type_of_Problem_not_set","save");
}

// Restores a previously saved state
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Precondition: initialize, not yet terminate, state has been saved
// label : an int identifying an already saved state
// method : the saving method
// Postcondition: The subsequent calls to any methods should give the exact
same result
// as if they were made after the corresponding call to save.
void Problem::restore(int label, const string& method) {
    throw NotImplemented("type_of_Problem_not_set","restore");
}

// Forgets a saved state
// Precondition: initialize, not yet terminate, state has been saved
// label : an int identifying an already saved state
// method : the saving method
// Postcondition: Memory/disk/... is freed as if the corresponding call to
save had not been made.
void Problem::forget(int label, const string& method) const {
    throw NotImplemented("type_of_Problem_not_set","forget");
}

// This method is used to find the names of input fields understood by the
Problem
// Precondition: initTimeStep
// Return value: list of names usable with getInputFieldTemplate and
setInputField
vector<string> Problem::getInputFieldsNames() const {
    throw NotImplemented("type_of_Problem_not_set","getInputFieldsNames");
    vector<string> v;
    return v;
}

// This method is used to get a template of a field expected for the given
name, i.e.
// a field similar to the one expected by setInputField with the same name.
// Precondition: initTimeStep, name is one of getInputFieldsNames
// name: the name of the input field
// afield: return value, a field similar to the one expected by setInputField
void Problem::getInputFieldTemplate(const string& name, TrioField& afield)
const {
    throw NotImplemented("type_of_Problem_not_set","getInputFieldTemplate");
}

// This method is used to provide the Problem with an input field.
// Precondition: initTimeStep, name is one of getInputFieldsNames, afield is
like in getInputFieldTemplate
// name: the name of the input field
// afield: the input field
// Postcondition: Values of afield have been used (copied inside the
Problem).
void Problem::setInputField(const string& name, const TrioField& afield) {
    throw NotImplemented("type_of_Problem_not_set","setInputField");
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
// This method is used to find the names of output fields understood by the
Problem
// It is not implemented and returns a void list (it would be very
// difficult to list all cathare computation variables...)
// Precondition: initTimeStep
// Return value: list of names usable with getOutputField
vector<string> Problem::getOutputFieldsNames() const {
    throw NotImplemented("type_of_Problem_not_set","getOutputFieldsNames");
    vector<string> v;
    return v;
}

// This method is used to get an output field for the given name
// Precondition: initTimeStep, name is one of getOutputFieldsNames
// name : the name of the output field
// afield: return value, the output field, with geometry and values filled
void Problem::getOutputField(const string& name, TrioField& afield) const {
    throw NotImplemented("type_of_Problem_not_set","getOutputField");
}

// This method is similar to getInputFieldTemplate but with a MEDCoupling
field
// Precondition: initTimeStep, name is one of getInputFieldsNames
// name: the name of the input field
// return value: a field similar to the one expected by setInputField
MEDCouplingFieldDouble* Problem::getInputMEDFieldTemplate(const string& name)
const {
    throw NotImplemented("type_of_Problem_not_set","getInputMEDFieldTemplate");
    return 0;
}

// This method is similar to setInputField but with a MEDCoupling field
// Precondition: initTimeStep, name is one of getInputFieldsNames, afield is
like in getInputFieldTemplate
// name: the name of the input field
// afield: the input field
// Postcondition: Values of afield have been used (copied inside the
Problem).
void Problem::setInputMEDField(const string& name, const
MEDCouplingFieldDouble* afield) {
    throw NotImplemented("type_of_Problem_not_set","setInputMEDField");
}

// This method is similar to getOutputField but with a MEDCoupling field
// Precondition: initTimeStep, name is one of getOutputFieldsNames
// name : the name of the output field
// return value: the output field, with geometry and values filled
MEDCouplingFieldDouble* Problem::getOutputMEDField(const string& name) const
{
    throw NotImplemented("type_of_Problem_not_set","getOutputMEDField");
    return 0;
}
```

*Documentation of the Interface for Code Coupling : ICoCo***5.4.3 Header file: ICoCoField.h**

```
#ifndef _ICoCoField_included_
#define _ICoCoField_included_
#include <string>

namespace ICoCo {
    class Field {
    public:
        Field();
        virtual ~Field();
        void setName(const std::string& name);
        const std::string& getName() const;
        const char* getCharName() const;

    private:
        std::string* _name;
    };
}
#endif
```

5.4.4 Source file: ICoCoField.cpp

```
#include <ICoCoField.h>
#include <string>

using namespace ICoCo;
using std::string;

Field::Field() {
    _name=new string;
}

Field::~Field() {
    delete _name;
}

void Field::setName(const string& name) {
    *_name=name;
}

const string& Field::getName() const {
    return *_name;
}

const char* Field::getCharName() const {
    return _name->c_str();
}
```


5.4.5 Header field: ICoCoTrioField.h

```

#ifndef _ICoCoTrioField_included_
#define _ICoCoTrioField_included_

#include <ICoCoField.h>
namespace ICoCo {
    // class TrioField, used for coupling Trio codes via the ICoCo interface
    // This structure contains all the necessary information
    // for constructing a ParaMEDMEM::ParaFIELD (with the addition of the MPI
    // communicator).
    // This structure can either own or not _field values
    (_has_field_ownership)
    // For _coords, _connectivity and _field, a null pointer means no data
    // allocated.
    // _coords and _connectivity tables, when allocated, are always owned by
the
    // TrioField.
    //
    class TrioField:public Field {
    public:

        TrioField();
        TrioField(const TrioField& OtherField);
        ~TrioField();
        void clear();
        void set_standalone();
        void dummy_geom();
        TrioField& operator=(const TrioField& NewField);
        void save(std::ostream& os) const;
        void restore(std::istream& in);
        int nb_values() const ;

    public:
        int _type ; // 0 elem 1 nodes
        int _mesh_dim;
        int _space_dim;
        int _nbnodes;
        int _nodes_per_elem;
        int _nb_elems;
        int _itnumber;
        int* _connectivity;
        double* _coords;

        double _time1,_time2;
        int _nb_field_components;
        double* _field;
        bool _has_field_ownership;
    };
}
#endif

```

*Documentation of the Interface for Code Coupling : ICoCo***5.4.6 Source file: ICoCoTrioField.cpp**

```

#include <ICoCoTrioField.h>
#include <string.h>
#include <iostream>
#include <iomanip>

using namespace ICoCo;
using namespace std;

// Default constructor
TrioField::TrioField() :
    _type(0),
    _mesh_dim(0),
    _space_dim(0),
    _nbnodes(0),
    _nodes_per_elem(0),
    _nb_elems(0),
    _itnumber(0),
    _connectivity(0),
    _coords(0),
    _time1(0.),
    _time2(0.),
    _nb_field_components(0),
    _field(0),
    _has_field_ownership(false) { }

// Copy constructor
TrioField::TrioField(const TrioField& OtherField) {
    (*this)=OtherField;
}

// Destructor
TrioField::~TrioField() {
    clear();
}

// After the call to clear(), all pointers are null and field ownership is
// false.
// Arrays are deleted if necessary
void TrioField::clear() {
    if (_connectivity)
        delete[] _connectivity;
    if (_coords)
        delete[] _coords;
    if (_field && _has_field_ownership)
        delete[] _field;
    _connectivity=0;
    _coords=0;
    _field=0;
    _has_field_ownership=false;
}

```

Documentation of the Interface for Code Coupling : ICoCo

```

// Returns the number of value locations
// The size of field is nb_values()*_nb_field_components
int TrioField::nb_values() const {
    if (_type==0)
        return _nb_elems;
    else if (_type==1)
        return _nbnodes;
    throw 0;
    //exit(-1);
    return -1;
}

// Save field to a .field file (loadable by visit!)
void TrioField::save(ostream& os) const{
    os << setprecision(12);
    os << getName() << endl;
    os << _type << endl;
    os << _mesh_dim << endl;
    os << _space_dim << endl;
    os << _nbnodes << endl;
    os << _nodes_per_elem << endl;
    os << _nb_elems << endl;

    os<< _itnumber<<endl;
    for (int i=0;i<_nb_elems;i++) {
        for (int j=0;j<_nodes_per_elem;j++)
            os << " " << _connectivity[i*_nodes_per_elem+j];
        os<<endl;
    }

    for (int i=0;i<_nbnodes;i++) {
        for (int j=0;j<_space_dim;j++)
            os << " " << _coords[i*_space_dim+j] ;
        os << endl;
    }

    os << _time1 << endl;
    os << _time2 << endl;
    os << _nb_field_components << endl;

    if (_field) {
        os << 1 << endl;
        for (int i=0;i<nb_values();i++) {
            for (int j=0;j<_nb_field_components;j++)
                os << " " << _field[i*_nb_field_components+j];
            os << endl;
        }
    }
    else
        os << 0 << endl;

    os << _has_field_ownership << endl;

```

Documentation of the Interface for Code Coupling : ICoCo

```
}

// Restore field from a .field file
void TrioField::restore(istream& in) {
    string name;
    in >> name;
    setName(name);
    in >> _type;
    in >> _mesh_dim;
    in >> _space_dim;
    in >> _nbnodes;
    in >> _nodes_per_elem;
    in >> _nb_elems;

    in >> _itnumber;
    if (_connectivity)
        delete [] _connectivity;
    _connectivity=new int[_nodes_per_elem*_nb_elems];
    for (int i=0;i<_nb_elems;i++) {
        for (int j=0;j<_nodes_per_elem;j++)
            in >> _connectivity[i*_nodes_per_elem+j];
    }
    if (_coords)
        delete [] _coords;
    _coords=new double[_nbnodes*_space_dim];
    for (int i=0;i<_nbnodes;i++) {
        for (int j=0;j<_space_dim;j++)
            in >> _coords[i*_space_dim+j];
    }

    in >> _time1;
    in >> _time2;
    in >> _nb_field_components;
    int test;
    in >> test;
    if (test) {
        if (_field)
            delete [] _field;
        _field=new double[_nb_field_components*nb_values()];
        for (int i=0;i<nb_values();i++) {
            for (int j=0;j<_nb_field_components;j++)
                in>> _field[i*_nb_field_components+j];
        }
    }
    else
        _field=0;

    in >> _has_field_ownership;
}

// After the call to set_standalone(), field ownership is true and field is
// allocated
// to the size _nb_field_components*nb_values().
```

Documentation of the Interface for Code Coupling : ICoCo

```
// The values of the field have been copied if necessary.
void TrioField::set_standalone() {
    if (!_field) {
        _field=new double[_nb_field_components*nb_values()];
        _has_field_ownership=true;
    }
    else if (!_has_field_ownership) {
        double *tmp_field=new double[_nb_field_components*nb_values()];
        memcpy(tmp_field,_field,_nb_field_components*nb_values()*sizeof(double));
        _field=tmp_field;
        _has_field_ownership=true;
    }
}

// Used to simulate a 0D geometry (Cathare/Trio for example).
void TrioField::dummy_geom() {
    _type=0;
    _mesh_dim=2;
    _space_dim=2;
    _nbnodes=3;
    _nodes_per_elem=3;
    _nb_elems=1;
    _itnumber=0;
    if (_connectivity)
        delete[] _connectivity;
    _connectivity=new int[3];
    _connectivity[0]=0;
    _connectivity[1]=1;
    _connectivity[2]=2;
    if (_coords)
        delete[] _coords;
    _coords=new double[6];
    _coords[0]=0;
    _coords[1]=0;
    _coords[2]=1;
    _coords[3]=0;
    _coords[4]=0;
    _coords[5]=1;
    _time1=0;
    _time2=1;
    _nb_field_components=1;
    if (_field && _has_field_ownership)
        delete[] _field;
    _has_field_ownership=false;
    _field=0;
}

// Overloading operator = for TrioField
// This becomes an exact copy of NewField.
// If NewField._has_field_ownership is false, they point to the same values.
// Otherwise the values are copied.
TrioField& TrioField::operator=(const TrioField& NewField){
    clear();
```

Documentation of the Interface for Code Coupling : ICoCo

```
_type=NewField._type;
_mesh_dim=NewField._mesh_dim;
_space_dim=NewField._space_dim;
_nbnodes=NewField._nbnodes;
_nodes_per_elem=NewField._nodes_per_elem;
_nb_elems=NewField._nb_elems;
_itnumber=NewField._itnumber;
_time1=NewField._time1;
_time2=NewField._time2;
_nb_field_components=NewField._nb_field_components;

if (!NewField._connectivity)
    _connectivity=0;
else {
    _connectivity=new int[_nodes_per_elem*_nb_elems];
    memcpy(
_connectivity,NewField._connectivity,_nodes_per_elem*_nb_elems*sizeof(int));
}

if (!NewField._coords)
    _coords=0;
else {
    _coords=new double[_nbnodes*_space_dim];
    memcpy( _coords,NewField._coords,_nbnodes*_space_dim*sizeof(double));
}

//Copie des valeurs du champ
_has_field_ownership=NewField._has_field_ownership;
if (_has_field_ownership) {
    _field=new double[nb_values()*_nb_field_components];

memcpy(_field,NewField._field,nb_values()*_nb_field_components*sizeof(double)
);
}
else
    _field=NewField._field;

return(*this);
}
```

*Documentation of the Interface for Code Coupling : ICoCo***5.4.7 Header file: Exceptions.h**

```

#include <exception>
#include <string>

#ifndef _Exceptions_included_
#define _Exceptions_included_

namespace ICoCo {
    class WrongContext : public std::exception {
    public:
        WrongContext(const std::string& prob, const std::string& method, const
std::string& precondition);
        ~WrongContext() throw() { }
        virtual const char* what() const throw();
    private:
        std::string prob;           // Problem in which exception occurred
        std::string method;        // method in which exception occurred
        std::string precondition; // precondition which was not met by the
Problem state
    };

    class WrongArgument : public std::exception {
    public:
        WrongArgument(const std::string& prob, const std::string& method, const
std::string& arg, const std::string& condition);
        ~WrongArgument() throw() { }
        virtual const char* what() const throw();
    private:
        std::string prob;           // Problem in which exception occurred
        std::string method;        // method in which exception occurred
        std::string arg;           // argument for which exception occurred
        std::string condition;     // condition which was not met by the argument
    };

    class NotImplemented : public std::exception {
    public:
        NotImplemented(const std::string& prob, const std::string& method);
        ~NotImplemented() throw() { }
        virtual const char* what() const throw();
    private:
        std::string prob;           // Problem in which exception occurred
        std::string method;        // method in which exception occurred
    };
}

#endif

```

5.4.8 Source file: Exceptions.cpp

```
#include <Exceptions.h>

#include <sstream>
using namespace std;

using namespace ICoCo;

WrongContext::WrongContext(const string& prob, const string& method, const
string& precondition) :
    prob(prob), method(method), precondition(precondition) { }

const char *WrongContext::what() const throw() {
    ostringstream s;
    s << "WrongContext in Problem instance of name: " << prob<< endl;
    s << " in method " << method << " : " << precondition << endl;
    return s.str().c_str();
}

WrongArgument::WrongArgument(const string& prob, const string& method, const
string& arg, const string& condition) :
    prob(prob), method(method), arg(arg), condition(condition) { }

const char *WrongArgument::what() const throw() {
    ostringstream s;
    s << "WrongContext in Problem instance of name: " << prob<< endl;
    s << " in method " << method << ", argument " << arg << " : " << condition
<< endl;
    return s.str().c_str();
}

NotImplemented::NotImplemented(const string& prob, const string& method) :
    prob(prob), method(method) { }

const char *NotImplemented::what() const throw() {
    ostringstream s;
    s << "WrongContext in Problem instance of name: " << prob<< endl;
    s << ", method " << method << endl;
    return s.str().c_str();
}
```


5.5 Annex 2: Main program of a sequential C++ supervisor

```
#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>

using namespace std;
using namespace ICoCo;

// OpenLib function
// Input parameter: the name of the library
// output parameter: a pointer to the problem
Problem* openLib(const char* libname, void* & handle) {
    // open shared library
    Problem *(*getProblem)();
    // open the code dynamic library
    handle =dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
        cerr << dlerror() << endl;
        throw 0;
    }
    // look at getProblem method in the code dynamic library
    getProblem=(Problem* (*)())dlsym(handle, "getProblem");
    if (!getProblem) {
        cout << dlerror() << endl;
        throw 0;
    }
    // instantiation of ICoCo Problem
    return (*getProblem)();
}

// closeLib function
// Input parameter : an opaque "handle" for the dynamic library
void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

// getFieldMean function
// input parameters: 1- pointer to ICoCo Problem which provides the output
// field
//                               2- name of the field to get
// output parameter: the mean of the field (a double)
double getFieldMean(Problem *P, string name) {
    double mean=0.;
    TrioField f ;

    // get the output field from the problem
```

Documentation of the Interface for Code Coupling : ICoCo

```
P->getOutputField(name,f);
// only one component fields are treated
if (f._nb_field_components!=1)
    throw 0;

// Compute and return the mean value (all elements/nodes have the same
weight)
for (int loc=0;loc<f.nb_values();loc++)
    mean=mean+f._field[loc];
return mean/f.nb_values();
}

// setConstantField function
// input parameters: 1- pointer to ICoCo Problem to which the field will be
set
//                      2- name of the field to set
//                      3- value to set
//                      4- component of the field to affect (1st one by default)
// output parameter: No output parameter

void setConstantField(Problem *P, string name, double val, int comp=0) {
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();

    // Fill the values
    // Give a constant field to a Problem.
    // One component of the field is specified, the others are 0.
    int nb_elems=f._nb_elems;
    int nb_comp=f._nb_field_components;
    for (int i=0;i<nb_elems;i++)
        for (int j=0;j<nb_comp;j++)
            if (j!=comp)
                f._field[i*nb_comp+j]=0;
            else
                f._field[i*nb_comp+j]=val;

    // Give the field to the problem
    P->setInputField(name,f);
}

int main(int argc, char** argv) {

    // open TRIO-U shared library and instantiate ICoCo Trio_U Problem T
    void* handle_trio;
    Problem *T=openLib("_Trio_UModule_opt.so", handle_trio);

    // open CATHARE shared library and instantiate ICoCo Cathare Problem C
    void* handle_cathare;
    Problem *C=openLib("./libcathare_gad.so", handle_cathare);
```

Documentation of the Interface for Code Coupling : ICoCo

```
// initialization of the problems
T->setDataFile('jdd_Trio.data');
T->setMPIComm(0) ; //sequential calculation
T->initialize() ;
C->initialize();

// Coupling parameters
// Time for which Cathare runs alone. Allows to reach a Cathare steady
state.
double t_begin_trio=1000;
// Time at which Trio_U -> Cathare retroaction on pressure begins
double time_retro_P=1001;
// Time at which Trio_U -> Cathare retroaction on temperature begins
double time_retro_T=1003;

// data to transform Cathare enthalpy into Trio_U temperature or vice versa
//  $H = h_{ref} + C_{p\_Trio} * (T - T_{ref})$ 
double href = 1.15116e6 ;
double tref = 263.8+273.15 ; // Trio_U temperature are in K
double Cp_Trio = 5332.43 ;

// mass flowrate transformation
// In Cathare, pipe of diameter 0.6m (S=0.28274m2)
// In Trio_U, 2D pipe of section 0.6m
// We keep the same  $Q = \rho * u * S$  between Cathare and Trio_U
// =>  $u(Trio) = Q(Cathare) / S(Trio) / \rho(Trio)$ 
// !!  $u(Trio) \neq u(Cathare)$ 
double S_Trio = 0.6 ;
double rho_Trio = 739.206 ;

// variables for pressure retroaction
// Must be remembered from timestep to timestep
double dP1=0; // outlet1 - inlet1
double dP2=0; // outlet2 - inlet1
double dP3=0; // inlet2 - inlet1

// First time loop (Cathare alone)
double epsilon=1e-10; // small time for double comparisons
bool stop; // Does the problem want to stop ?

while (1) { // Loop on timesteps
    double present=C->presentTime();
    double dt=C->computeTimeStep(stop);
    if (stop || present>t_begin_trio-epsilon)
        break;

    // Modify dt in order to come to exactly t_begin_trio
    if (present+dt>t_begin_trio)
        dt=t_begin_trio-present;
    else if (present+2*dt>t_begin_trio)
        dt=(t_begin_trio-present)/2;
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Initialize the timestep
C->initTimeStep(dt);

// Perform the computation
bool ok=C->solveTimeStep();

// Either validate or abort it
if (!ok) // The resolution failed, try with a new dt
    C->abortTimeStep();
else // Validate and go to the next time step
    C->validateTimeStep();

} // End loop on timesteps

// Second time loop (both codes)
bool ok=true; // Is the time interval successfully solved?
stop=false;

while (!stop) { // Loop on timesteps
    ok=false;
    while (!ok) { // Loop on timestep size
        bool stop_T,stop_C;

        // Compute time step length
        double dt_C=C->computeTimeStep(stop_C);
        double dt_T=T->computeTimeStep(stop_T);
        double dt=min(dt_C,dt_T);

        // And decide if we have to stop
        stop = stop_T || stop_C;
        if (stop)
            break;

        //prepare the new time step
        C->initTimeStep(dt);
        T->initTimeStep(dt);

        // Get output from Cathare
        double C_pressure_inlet1, C_pressure_outlet1;
        double C_pressure_inlet2, C_pressure_outlet2;
        double C_massflow_inlet1, C_massflow_outlet1;
        double C_massflow_inlet2, C_massflow_outlet2;
        double C_enthalpy_inlet1, C_enthalpy_outlet1;
        double C_enthalpy_inlet2, C_enthalpy_outlet2;

        // Pressures
        C_pressure_inlet1=getFieldMean(C,"PRESSURE_PIPE11_85");
        C_pressure_outlet1=getFieldMean(C,"PRESSURE_PIPE12_1");
        C_pressure_inlet2=getFieldMean(C,"PRESSURE_PIPE21_85");
        C_pressure_outlet2=getFieldMean(C,"PRESSURE_PIPE22_1");

        // Mass flow rates
```

Documentation of the Interface for Code Coupling : ICoCo

```

C_massflow_inlet1=getFieldMean(C,"LIQFLOW_PIPE11_85");
C_massflow_outlet1=getFieldMean(C,"LIQFLOW_PIPE12_1");
C_massflow_inlet2=getFieldMean(C,"LIQFLOW_PIPE21_85");
C_massflow_outlet2=getFieldMean(C,"LIQFLOW_PIPE22_1");

// Enthalpies
C_enthalpy_inlet1=getFieldMean(C,"LIQH_PIPE11_85");
C_enthalpy_outlet1=getFieldMean(C,"LIQH_PIPE12_1");
C_enthalpy_inlet2=getFieldMean(C,"LIQH_PIPE21_85");
C_enthalpy_outlet2=getFieldMean(C,"LIQH_PIPE22_1");

// Compute Trio_U input values
//calculation of input temperature for Trio_U
double temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref;
double temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref;
double temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref;
double temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref;

// calculation of inlet and outlet velocity for Trio_U
double u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio;
double u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio;
double u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio;
double u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio;

// Modifications to ensure div(u)=0
// works because all areas are equal.
double divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2;
u_outlet1-=divergence/4;
u_outlet2-=divergence/4;
u_inlet1+=divergence/4;
u_inlet2+=divergence/4;

// Get output from Trio_U
double T_pressure_inlet1, T_pressure_inlet2 ;
double T_pressure_outlet1, T_pressure_outlet2;
double T_temperature_inlet1, T_temperature_inlet2 ;
double T_temperature_outlet1, T_temperature_outlet2;

// Pressures
T_pressure_inlet1=getFieldMean(T,"pressure_inlet1")*rho_Trio;
T_pressure_inlet2=getFieldMean(T,"pressure_inlet2")*rho_Trio;
T_pressure_outlet1=getFieldMean(T,"pressure_outlet1")*rho_Trio;
T_pressure_outlet2=getFieldMean(T,"pressure_outlet2")*rho_Trio;

// Temperatures
T_temperature_inlet1=getFieldMean(T,"temperature_inlet1");
T_temperature_inlet2=getFieldMean(T,"temperature_inlet2");
T_temperature_outlet1=getFieldMean(T,"temperature_outlet1");
T_temperature_outlet2=getFieldMean(T,"temperature_outlet2");

// Give input to Cathare
// Trio_U -> Cathare retroaction on pressure
if (C->presentTime(>time_retro_P) {

```

Documentation of the Interface for Code Coupling : ICoCo

```

double dP_T1 = T_pressure_inlet1-T_pressure_outlet1;
double dP_T2 = T_pressure_inlet1-T_pressure_outlet2;
double dP_T3 = T_pressure_inlet1-T_pressure_inlet2;

double dP_C1 = C_pressure_inlet1-C_pressure_outlet1;
double dP_C2 = C_pressure_inlet1-C_pressure_outlet2;
double dP_C3 = C_pressure_inlet1-C_pressure_inlet2;

double tau_coupl=0.1;
double k_loc=0;
if (dt>tau_coupl)
    k_loc=1;
else
    k_loc=dt/tau_coupl;

// pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc;
dP2 += (dP_C2-dP_T2)*k_loc;
dP3 -= (dP_C3-dP_T3)*k_loc;

setConstantField(C,"DPLEXT_PIPE12_1",dP1);
setConstantField(C,"DPLEXT_PIPE22_1",dP2);
setConstantField(C,"DPLEXT_PIPE21_86",dP3);
}

// Trio_U -> Cathare retroaction on temperature
if (C->presentTime()>time_retro_T) {
    double T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref);
    double T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref);
    double T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref);
    double T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref);

    // mesh numbers are growing from "left" to "right"
    // overlapping domain on the "right"
    // Loop 1
    setConstantField(C,"OVHLEXT_PIPE11_85",-10);
    setConstantField(C,"OVPLEXT_PIPE11_85",-10);
    setConstantField(C,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1);
    // Loop 2
    setConstantField(C,"OVHLEXT_PIPE21_85",-10);
    setConstantField(C,"OVPLEXT_PIPE21_85",-10);
    setConstantField(C,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2);

    // overlapping domain on the "left"
    // Loop 1
    setConstantField(C,"OVHLEXT_PIPE12_1",10);
    setConstantField(C,"OVPLEXT_PIPE12_2",10);
    setConstantField(C,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1);
    // Loop 2
    setConstantField(C,"OVHLEXT_PIPE22_1",10);
    setConstantField(C,"OVPLEXT_PIPE22_2",10);
    setConstantField(C,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2);
}

```

Documentation of the Interface for Code Coupling : ICoCo

```
// Give input to Trio_U
// Velocities
setConstantField(T,"u_inlet1",u_inlet1,1);
setConstantField(T,"u_inlet2",u_inlet2,1);
setConstantField(T,"u_outlet1",u_outlet1,1);
setConstantField(T,"u_outlet2",u_outlet2,1);

// Temperatures
setConstantField(T,"temperature_inlet1",temperature_inlet1);
setConstantField(T,"temperature_inlet2",temperature_inlet2);
setConstantField(T,"temperature_outlet1",temperature_outlet1);
setConstantField(T,"temperature_outlet2",temperature_outlet2);

// Solve next time step
bool ok_T=T->solveTimeStep();
bool ok_C=C->solveTimeStep();
ok = ok_T && ok_C;

// Two ways : The resolution failed, try with a new dt or validate and
go // to next time step
if (!ok) {
    T->abortTimeStep();
    C->abortTimeStep();
    cout << "Abort at time " << C-> presentTime() << endl;
}
else {
    T->validateTimeStep();
    C->validateTimeStep();
}
} // End loop on timestep size
} // End loop on timesteps

T->terminate();
C->terminate();

delete T;
delete C;

closeLib(handle_cathare);
closeLib(handle_trio);

return 0;
}
```

5.6 Annex 3: Main program of a parallel C++ supervisor

```
#include <Problem.h>
#include <ICoCoTrioField.h>
#include <dlfcn.h>
#include <iostream>
#include <sstream>
#include <mpi.h>

using namespace std;
using namespace ICoCo;

// Get min, max, and, or of variables on all processes of MPI_COMM_WORLD
int mpi_min(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
int mpi_max(int i) {
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
double mpi_min(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
    return result;
}
double mpi_max(double i) {
    double result;
    MPI_Allreduce(&i, &result, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
bool mpi_and(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    return (result != 0);
}
bool mpi_or(bool b) {
    int i = b ? 1 : 0;
    int result;
    MPI_Allreduce(&i, &result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}

// Get the sum on all processes of comm
// To be called on all processes of comm
double mpi_sum(double d, MPI_Comm comm) {
    double result;
```


Documentation of the Interface for Code Coupling : ICoCo

```

    MPI_Allreduce(&d, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
    return result;
}

// OpenLib function
// Input parameter: the name of the library
// output parameter: a pointer to the problem
//                               : an opaque "handle" for the dynamic library
Problem* openLib(const char* libname, void* & handle) {
    // open shared library
    Problem *(*getProblem)();
    // open the code dynamic library
    handle =dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
        cerr << dlerror() << endl;
        throw 0;
    }
    // look at getProblem method in the code dynamic library
    getProblem=(Problem* (*)())dlsym(handle, "getProblem");
    if (!getProblem) {
        cout << dlerror() << endl;
        throw 0;
    }
    // instantiation of ICoCo Problem
    return (*getProblem)();
}

// closeLib function
// Input parameter : an opaque "handle" for the dynamic library
void closeLib(void* handle) {
    if (dlclose(handle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

// getFieldMean function
// input parameters: 1- pointer to ICoCo Problem which provides the output
// field
//                               2- name of the field to get
//                               3- MPI_Communicator (equal to 0 if only one process)
// output parameter: the mean of the field (a double)
double getFieldMean(Problem *P, string name, MPI_Comm comm=0) {
    double mean=0.;
    TrioField f ;

    // get the output field from the problem
    P->getOutputField(name,f);
    // only one component fields are treated
    if (f._nb_field_components!=1)
        throw 0;
}

```

Documentation of the Interface for Code Coupling : ICoCo

```
// Compute and return the mean value (all elements/nodes have the same
weight)
for (int loc=0;loc<f.nb_values();loc++)
    mean=mean+f._field[loc];

double nb_values=f.nb_values();
if (comm) {
    mean=mpi_sum(mean,comm);
    nb_values=mpi_sum(nb_values,comm);
}

return mean/nb_values;
}

// setConstantField function
// input parameters: 1- pointer to ICoCo Problem to which the field will be
set
//
//           2- name of the field to set
//           3- value to set
//           4- component of the field to affect (1st one by default)
// output parameter: No output parameter

void setConstantField(Problem *P, string name, double val, int comp=0) {
    TrioField f;
    // Get the right geometry and format for the field
    P->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();

    // Fill the values
    // Give a constant field to a Problem.
    // One component of the field is specified, the others are 0.
    int nb_elems=f._nb_elems;
    int nb_comp=f._nb_field_components;
    for (int i=0;i<nb_elems;i++)
        for (int j=0;j<nb_comp;j++)
            if (j!=comp)
                f._field[i*nb_comp+j]=0;
            else
                f._field[i*nb_comp+j]=val;

    // Give the field to the problem
    P->setInputField(name,f);
}

int main(int argc, char** argv) {

// MPI initialization
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Documentation of the Interface for Code Coupling : ICoCo

```

// Create the communicators for Cathare and Trio_U
int trio_ids[size-1];
for (int i = 0; i < size - 1; i++)
    trio_ids[i] = i + 1;
int cath_ids[1];
    cath_ids[0] = 0;
MPI_Group group_world;
MPI_Group group_trio;
MPI_Comm comm_trio;
// Create the group world
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
// Create the group trio
MPI_Group_incl(group_world, size - 1, trio_ids, &group_trio);
// Create the communicator trio
MPI_Comm_create(MPI_COMM_WORLD, group_trio, &comm_trio);

// Choose if this process handles Cathare or Trio_U
// the first processor for Cathare, all the others for Trio_U
string progname;
string libname;
if (rank == 0) {
    progname = "Cathare";
    libname = "./libcathare_gad.so";
} else {
    progname = "Trio";
    libname = "_Trio_UModule_opt.so";
}

// Redirect the outputs
ostringstream out;
out << progname << ".out" << ends;
ostringstream err;
err << progname << ".err" << ends;

freopen(out.str().c_str(), "w", stdout);
freopen(err.str().c_str(), "w", stderr);

// Open dynamic libraries
void* handle;
Problem *P=openLib(libname.c_str(), handle);

// initialization of the problems
// Cathare doesn't need any initialization parameter
// Trio_U needs both a datafile name and an MPI communicator
if (progname=="Trio") {
    if (size<3)
        P->setDataFile("jdd_Trio.data");
    else
        P->setDataFile("PAR_jdd_Trio.data");
    P->setMPIComm(&comm_trio);
}
P->initialize();

```

Documentation of the Interface for Code Coupling : ICoCo

```

// Coupling parameters
// Time for which Cathare runs alone. Allows to reach a Cathare steady
state.
double t_begin_trio=1000;
// Time at which Trio_U -> Cathare retroaction on pressure begins
double time_retro_P=1001;
// Time at which Trio_U -> Cathare retroaction on temperature begins
double time_retro_T=1003;

// data to transform Cathare enthalpy into Trio_U temperature or vice versa
//  $H = h_{ref} + C_p \cdot (T - T_{ref})$ 
double href = 1.15116e6 ;
double tref = 263.8+273.15 ; // Trio_U temperature are in K
double Cp_Trio = 5332.43 ;

// mass flowrate transformation
// In Cathare, pipe of diameter 0.6m ( $S=0.28274m^2$ )
// In Trio_U, 2D pipe of section 0.6m
// We keep the same  $Q=\rho \cdot u \cdot S$  between Cathare and Trio_U
// =>  $u(Trio) = Q(Cathare) / S(Trio) / \rho(Trio)$ 
// !!  $u(Trio) \neq u(Cathare)$ 
double S_Trio = 0.6 ;
double rho_Trio = 739.206 ;

// variables for pressure retroaction
// Must be remembered from timestep to timestep
double dP1=0; // outlet1 - inlet1
double dP2=0; // outlet2 - inlet1
double dP3=0; // inlet2 - inlet1

// First time loop (Cathare alone)
if (programe=="Cathare") {
    double epsilon=1e-10; // small time for double comparisons
    bool stop; // Does the problem want to stop ?

    while (1) { // Loop on timesteps
        double present=P->presentTime();
        double dt=P->computeTimeStep(stop);
        if (stop || present>t_begin_trio-epsilon)
            break;

        // Modify dt in order to come to exactly t_begin_trio
        if (present+dt>t_begin_trio)
            dt=t_begin_trio-present;
        else if (present+2*dt>t_begin_trio)
            dt=(t_begin_trio-present)/2;

        // Initialize the timestep
        P->initTimeStep(dt);

        // Perform the computation
        bool ok=P->solveTimeStep();
    }
}

```

Documentation of the Interface for Code Coupling : ICoCo

```

// Either validate or abort it
if (!ok) // The resolution failed, try with a new dt
    P->abortTimeStep();
else // Validate and go to the next time step
    P->validateTimeStep();
} // End loop on timesteps
}

// Second time loop (both codes)
bool ok=true; // Is the time interval successfully solved?
bool stop=false;

while (!stop) { // Loop on timesteps
    ok=false;
    while (!ok) { // Loop on timestep size
        // Compute time step length
        double dt =P->computeTimeStep(stop);
        dt=mpi_min(dt);
        // And decide if we have to stop
        stop = mpi_or(stop);
        if (stop)
            break;

        //prepare the new time step
        P->initTimeStep(dt);

        // Get output from Cathare
        double C_pressure_inlet1, C_pressure_outlet1;
        double C_pressure_inlet2, C_pressure_outlet2;
        double C_massflow_inlet1, C_massflow_outlet1;
        double C_massflow_inlet2, C_massflow_outlet2;
        double C_enthalpy_inlet1, C_enthalpy_outlet1;
        double C_enthalpy_inlet2, C_enthalpy_outlet2;

        if (programe=="Cathare") {
            // Pressures
            C_pressure_inlet1=getFieldMean(P,"PRESSURE_PIPE11_85");
            C_pressure_outlet1=getFieldMean(P,"PRESSURE_PIPE12_1");
            C_pressure_inlet2=getFieldMean(P,"PRESSURE_PIPE21_85");
            C_pressure_outlet2=getFieldMean(P,"PRESSURE_PIPE22_1");

            // Mass flow rates
            C_massflow_inlet1=getFieldMean(P,"LIQFLOW_PIPE11_85");
            C_massflow_outlet1=getFieldMean(P,"LIQFLOW_PIPE12_1");
            C_massflow_inlet2=getFieldMean(P,"LIQFLOW_PIPE21_85");
            C_massflow_outlet2=getFieldMean(P,"LIQFLOW_PIPE22_1");

            // Enthalpies
            C_enthalpy_inlet1=getFieldMean(P,"LIQH_PIPE11_85");
            C_enthalpy_outlet1=getFieldMean(P,"LIQH_PIPE12_1");
            C_enthalpy_inlet2=getFieldMean(P,"LIQH_PIPE21_85");

```

Documentation of the Interface for Code Coupling : ICoCo

```

    C_enthalpy_outlet2=getFieldMean(P,"LIQH_PIPE22_1");
}

// Transfer the data to Trio_U
MPI_Bcast(&C_pressure_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_pressure_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Bcast(&C_massflow_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_massflow_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_inlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_outlet1, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_inlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&C_enthalpy_outlet2, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

// Compute Trio_U input values
//calculation of input temperature for Trio_U
double temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref;
double temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref;
double temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref;
double temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref;

// calculation of inlet and outlet velocity for Trio_U
double u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio;
double u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio;
double u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio;
double u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio;

// Modifications to ensure div(u)=0
// works because all areas are equal.
double divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2;
u_outlet1-=divergence/4;
u_outlet2-=divergence/4;
u_inlet1+=divergence/4;
u_inlet2+=divergence/4;

// Get output from Trio_U
double T_pressure_inlet1, T_pressure_inlet2 ;
double T_pressure_outlet1, T_pressure_outlet2;
double T_temperature_inlet1, T_temperature_inlet2 ;
double T_temperature_outlet1, T_temperature_outlet2;

if (programe=="Trio") {
    // Pressures

T_pressure_inlet1=getFieldMean(P,"pressure_inlet1",comm_trio)*rho_Trio;

T_pressure_inlet2=getFieldMean(P,"pressure_inlet2",comm_trio)*rho_Trio;

T_pressure_outlet1=getFieldMean(P,"pressure_outlet1",comm_trio)*rho_Trio;

```

Documentation of the Interface for Code Coupling : ICoCo

```

T_pressure_outlet2=getFieldMean(P,"pressure_outlet2",comm_trio)*rho_Trio;

// Temperatures
T_temperature_inlet1=getFieldMean(P,"temperature_inlet1",comm_trio);
T_temperature_inlet2=getFieldMean(P,"temperature_inlet2",comm_trio);

T_temperature_outlet1=getFieldMean(P,"temperature_outlet1",comm_trio);

T_temperature_outlet2=getFieldMean(P,"temperature_outlet2",comm_trio);
}

// Transfer the data to Cathare
MPI_Bcast(&T_pressure_inlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_outlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_inlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_pressure_outlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);

MPI_Bcast(&T_temperature_inlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_inlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_outlet1, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);
MPI_Bcast(&T_temperature_outlet2, 1, MPI_DOUBLE, 1, MPI_COMM_WORLD);

// Give input to Cathare
// Trio_U -> Cathare retroaction on pressure
if (programe=="Cathare") {
    if (P->presentTime()>time_retro_P) {
        double dP_T1 = T_pressure_inlet1-T_pressure_outlet1;
        double dP_T2 = T_pressure_inlet1-T_pressure_outlet2;
        double dP_T3 = T_pressure_inlet1-T_pressure_inlet2;

        double dP_C1 = C_pressure_inlet1-C_pressure_outlet1;
        double dP_C2 = C_pressure_inlet1-C_pressure_outlet2;
        double dP_C3 = C_pressure_inlet1-C_pressure_inlet2;

        double tau_coupl=0.1;
        double k_loc=0;
        if (dt>tau_coupl)
            k_loc=1;
        else
            k_loc=dt/tau_coupl;

        // pressure retroaction on outlets
        dP1 += (dP_C1-dP_T1)*k_loc;
        dP2 += (dP_C2-dP_T2)*k_loc;
        dP3 -= (dP_C3-dP_T3)*k_loc;

        setConstantField(P,"DPLEXT_PIPE12_1",dP1);
        setConstantField(P,"DPLEXT_PIPE22_1",dP2);
        setConstantField(P,"DPLEXT_PIPE21_86",dP3);
    }
}

// Trio_U -> Cathare retroaction on temperature

```

Documentation of the Interface for Code Coupling : ICoCo

```

if (P->presentTime()>time_retro_T) {
    double T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref);
    double T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref);
    double T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-
tref);
    double T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-
tref);

    // mesh numbers are growing from "left" to "right"
    // overlapping domain on the "right"
    // Loop 1
    setConstantField(P,"OVHLEXT_PIPE11_85",-10);
    setConstantField(P,"OVPLEXT_PIPE11_85",-10);
    setConstantField(P,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1);
    // Loop 2
    setConstantField(P,"OVHLEXT_PIPE21_85",-10);
    setConstantField(P,"OVPLEXT_PIPE21_85",-10);
    setConstantField(P,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2);

    // overlapping domain on the "left"
    // Loop 1
    setConstantField(P,"OVHLEXT_PIPE12_1",10);
    setConstantField(P,"OVPLEXT_PIPE12_2",10);
    setConstantField(P,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1);
    // Loop 2
    setConstantField(P,"OVHLEXT_PIPE22_1",10);
    setConstantField(P,"OVPLEXT_PIPE22_2",10);
    setConstantField(P,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2);
}
}
// Give input to Trio_U
if (programe=="Trio") {
    // Velocities
    setConstantField(P,"u_inlet1",u_inlet1,1);
    setConstantField(P,"u_inlet2",u_inlet2,1);
    setConstantField(P,"u_outlet1",u_outlet1,1);
    setConstantField(P,"u_outlet2",u_outlet2,1);

    // Temperatures
    setConstantField(P,"temperature_inlet1",temperature_inlet1);
    setConstantField(P,"temperature_inlet2",temperature_inlet2);
    setConstantField(P,"temperature_outlet1",temperature_outlet1);
    setConstantField(P,"temperature_outlet2",temperature_outlet2);
}
// Solve next time step
bool ok =P->solveTimeStep();
ok =mpi_and(ok) ;

// Two ways : The resolution failed, try with a new dt or validate and
go
// to next time step
if (!ok) {

```


Documentation of the Interface for Code Coupling : ICoCo

```
P->abortTimeStep();
cout << "Abort at time " << P-> presentTime() << endl;
}
else {
    P->validateTimeStep();
}
}
// End loop on timestep size
}
// End loop on timesteps

P->terminate();

delete P;
closeLib(handle);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();

return 0;
}
```

5.7 Annex 4: python sequential supervisor

```

import ICoCoModule,os
from ICoCoModule import getFieldMean,setConstantField,openLib, closeLib

# open shared libraries
T,handle_Trio=openLib("_Trio_UModule_opt.so")
C,handle_Cathare=openLib("./libcathare_gad.so")

# Initialize the problems
T.setDataFile("jdd_Trio.data")
T.initialize()

C.initialize()

# Coupling parameters
# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
# H = href + Cp_Trio * ( T - Tref )
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m (S=0.28274m2)
# In Trio_U, 2D pipe of section 0.6m
# We keep the same Q=rho*u*S between Cathare and Trio_U
# => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
# !! u(Trio) != u(Cathare)
S_Trio = 0.6
rho_Trio = 739.206

# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1

# First time loop (Cathare alone)
epsilon=1e-10 # small time for double comparisons

while (1) : # Loop on timesteps
    present=C.presentTime()
    dt, stop =C.computeTimeStep(stop_p)
    if (stop or present>t_begin_trio-epsilon):

```

Documentation of the Interface for Code Coupling : ICoCo

```
break

# Modify dt in order to come to exactly t_begin_trio
if (present+dt>t_begin_trio):
    dt=t_begin_trio-present
elif (present+2*dt>t_begin_trio):
    dt=(t_begin_trio-present)/2

# Initialize the timestep
C.initTimeStep(dt)

# Perform the computation
ok=C.solveTimeStep()

# Either validate or abort it
if not(ok): # The resolution failed, try with a new dt
    C.abortTimeStep()
else:      # Validate and go to the next time step
    C.validateTimeStep()
pass      # End loop on timesteps

# Second time loop (both codes)
ok=true   # Is the time interval successfully solved?
stop=False

while not(stop) : # Loop on timesteps
    ok=False
    while not(ok) : # Loop on timestep size

        # Compute time step length
        dt_C, stop_C=C.computeTimeStep(stop_C_p)
        dt_T, stop_T=T.computeTimeStep(stop_T_p)
        dt=min(dt_C,dt_T)

        # And decide if we have to stop
        stop = stop_T or stop_C
        if (stop):
            break

        # prepare the new time step
        C.initTimeStep(dt)
        T.initTimeStep(dt)

        # Get output from Cathare
        # Pressures
        C_pressure_inlet1=getFieldMean(C,"PRESSURE_PIPE11_85")
        C_pressure_outlet1=getFieldMean(C,"PRESSURE_PIPE12_1")
        C_pressure_inlet2=getFieldMean(C,"PRESSURE_PIPE21_85")
        C_pressure_outlet2=getFieldMean(C,"PRESSURE_PIPE22_1")

        # Mass flow rates
        C_massflow_inlet1=getFieldMean(C,"LIQFLOW_PIPE11_85")
        C_massflow_outlet1=getFieldMean(C,"LIQFLOW_PIPE12_1")
```

Documentation of the Interface for Code Coupling : ICoCo

```

C_massflow_inlet2=getFieldMean(C,"LIQFLOW_PIPE21_85")
C_massflow_outlet2=getFieldMean(C,"LIQFLOW_PIPE22_1")

# Enthalpies
C_enthalpy_inlet1=getFieldMean(C,"LIQH_PIPE11_85")
C_enthalpy_outlet1=getFieldMean(C,"LIQH_PIPE12_1")
C_enthalpy_inlet2=getFieldMean(C,"LIQH_PIPE21_85")
C_enthalpy_outlet2=getFieldMean(C,"LIQH_PIPE22_1")

# Compute Trio_U input values
#calculation of input temperature for Trio_U
temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref

# calculation of inlet and outlet velocity for Trio_U
u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio

# Modifications to ensure div(u)=0
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4

# Get output from Trio_U

# Pressures
T_pressure_inlet1=getFieldMean(T,"pressure_inlet1")*rho_Trio
T_pressure_inlet2=getFieldMean(T,"pressure_inlet2")*rho_Trio
T_pressure_outlet1=getFieldMean(T,"pressure_outlet1")*rho_Trio
T_pressure_outlet2=getFieldMean(T,"pressure_outlet2")*rho_Trio

# Temperatures
T_temperature_inlet1=getFieldMean(T,"temperature_inlet1")
T_temperature_inlet2=getFieldMean(T,"temperature_inlet2")
T_temperature_outlet1=getFieldMean(T,"temperature_outlet1")
T_temperature_outlet2=getFieldMean(T,"temperature_outlet2")

# Give input to Cathare
# Trio_U -> Cathare retroaction on pressure
if (C.presentTime(>time_retro_P) :
    dP_T1 = T_pressure_inlet1-T_pressure_outlet1
    dP_T2 = T_pressure_inlet1-T_pressure_outlet2
    dP_T3 = T_pressure_inlet1-T_pressure_inlet2

    dP_C1 = C_pressure_inlet1-C_pressure_outlet1
    dP_C2 = C_pressure_inlet1-C_pressure_outlet2

```

Documentation of the Interface for Code Coupling : ICoCo

```
dP_C3 = C_pressure_inlet1-C_pressure_inlet2

tau_coupl=0.1
k_loc=0
if (dt>tau_coupl):
    k_loc=1
else:
    k_loc=dt/tau_coupl

# pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc
dP2 += (dP_C2-dP_T2)*k_loc
dP3 -= (dP_C3-dP_T3)*k_loc

setConstantField(C,"DPLEXT_PIPE12_1",dP1)
setConstantField(C,"DPLEXT_PIPE22_1",dP2)
setConstantField(C,"DPLEXT_PIPE21_86",dP3)
pass

# Trio_U -> Cathare retroaction on temperature
if (C.presentTime()>time_retro_T) :
    T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
    T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
    T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
    T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)

    # mesh numbers are growing from "left" to "right"
    # overlapping domain on the "right"
    # Loop 1
    setConstantField(C,"OVHLEXT_PIPE11_85",-10)
    setConstantField(C,"OVPLEXT_PIPE11_85",-10)
    setConstantField(C,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1)
    # Loop 2
    setConstantField(C,"OVHLEXT_PIPE21_85",-10)
    setConstantField(C,"OVPLEXT_PIPE21_85",-10)
    setConstantField(C,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2)

    # overlapping domain on the "left"
    # Loop 1
    setConstantField(C,"OVHLEXT_PIPE12_1",10)
    setConstantField(C,"OVPLEXT_PIPE12_2",10)
    setConstantField(C,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1)
    # Loop 2
    setConstantField(C,"OVHLEXT_PIPE22_1",10)
    setConstantField(C,"OVPLEXT_PIPE22_2",10)
    setConstantField(C,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2)
    pass

# Give input to Trio_U
# Velocities
setConstantField(T,"u_inlet1",u_inlet1,1)
setConstantField(T,"u_inlet2",u_inlet2,1)
setConstantField(T,"u_outlet1",u_outlet1,1)
```

Documentation of the Interface for Code Coupling : ICoCo

```
setConstantField(T,"u_outlet2",u_outlet2,1)
```

Temperatures

```
setConstantField(T,"temperature_inlet1",temperature_inlet1)
setConstantField(T,"temperature_inlet2",temperature_inlet2)
setConstantField(T,"temperature_outlet1",temperature_outlet1)
setConstantField(T,"temperature_outlet2",temperature_outlet2)
```

Solve next time step

```
ok_T=T.solveTimeStep()
ok_C=C.solveTimeStep()
ok = ok_T and ok_C
```

Two ways : The resolution failed, try with a new dt or validate and go to next time step

```
if not(ok) :
    T.abortTimeStep()
    C.abortTimeStep()
    print "Abort at time " + C.presentTime()
else:
    T.validateTimeStep()
    C.validateTimeStep()
```

```
pass # End loop on timestep size
pass # End loop on timesteps
```

```
T.terminate()
C.terminate()
closeLib(handle_Trio)
closeLib(handle_Cathare)
```

5.8 Annex 5: python parallel supervisor

```
import ICoCoModule,os
from ICoCoModule import getFieldMean,setConstantField,openLib, closeLib
from ICoCoModule import mpi_min,mpi_or, mpi_bcast, mpi_and, my_MPI_Comm_size,
my_MPI_Comm_rank,MPI_Comm_new , my_MPI_Init

import sys

my_MPI_Init(sys.argv)

size = my_MPI_Comm_size()
rank = my_MPI_Comm_rank()

trio_ids = range(1,size)
comm_trio = MPI_Comm_new(size-1, trio_ids)

if rank == 0:
    progname="Cathare"
    libname = "./libcathare_gad.so"
    pass
else:
    progname = "Trio"
    libname =
"/home/edeville/CATHARE/ICOCO/Gauthier/ICoCo/build/exec_opt/_Trio_UModule_opt
.so"
    pass

prog_out = progname + ".out"
sys.stdout = open(prog_out,"w")
prog_err = progname + ".err"
sys.stderr = open(prog_err,"w")

# open shared libraries
P, handle =openLib(libname)

# Initialize the problems
if (progname=="Trio"):
    if (size<3):
        P.setDataFile("jdd_Trio.data")
        pass
    else:
        P.setDataFile("PAR_jdd_Trio.data")
        pass
    P.setMPIComm(comm_trio)
    pass
P.initialize()

# Coupling parameters
```

Documentation of the Interface for Code Coupling : ICoCo

```

# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
# H = href + Cp_Trio * ( T - Tref )
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m (S=0.28274m2)
# In Trio_U, 2D pipe of section 0.6m
# We keep the same Q=rho*u*S between Cathare and Trio_U
# => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
# !! u(Trio) != u(Cathare)
S_Trio = 0.6
rho_Trio = 739.206

# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1

# First time loop (Cathare alone)
epsilon=1e-10 # small time for double comparisons

if (programe=="Cathare"):
    while (1) : # Loop on timesteps
        present=P.presentTime()
        dt, stop =P.computeTimeStep()
        #stop = stop_p.value()
        if (stop or present>t_begin_trio-epsilon):
            break

        # Modify dt in order to come to exactly t_begin_trio
        if (present+dt>t_begin_trio):
            dt=t_begin_trio-present
        elif (present+2*dt>t_begin_trio):
            dt=(t_begin_trio-present)/2

        # Initialize the timestep
        P.initTimeStep(dt)

        # Perform the computation
        ok=P.solveTimeStep()

        # Either validate or abort it
        if not(ok): # The resolution failed, try with a new dt

```


Documentation of the Interface for Code Coupling : ICoCo

```
P.abortTimeStep()
else:      # Validate and go to the next time step
    P.validateTimeStep()
    pass                                       # End loop on timesteps
pass
pass

# Second time loop (both codes)
ok=True   # Is the time interval successfully solved?
stop=False
while not(stop) :                            # Loop on timesteps
    ok=False
    while not(ok) :                           # Loop on timestep size

        # Compute time step length
        dt, stop =P.computeTimeStep()
        dt=mpi_min(dt)
        stop=mpi_or(stop)

        # And decide if we have to stop
        if (stop):
            break

        # prepare the new time step
        P.initTimeStep(dt)

        # Get output from Cathare
        # Pressures
        C_pressure_inlet1 = 0.
        C_pressure_outlet1 = 0.
        C_pressure_inlet2 = 0.
        C_pressure_outlet2 = 0.
        C_massflow_inlet1 = 0.
        C_massflow_outlet1 = 0.
        C_massflow_inlet2 = 0.
        C_massflow_outlet2 = 0.
        C_enthalpy_inlet1 = 0.
        C_enthalpy_outlet1 = 0.
        C_enthalpy_inlet2 = 0.
        C_enthalpy_outlet2 = 0.
        if (progname=="Cathare") :
            C_pressure_inlet1=getFieldMean(P, "PRESSURE_PIPE11_85")
            C_pressure_outlet1=getFieldMean(P, "PRESSURE_PIPE12_1")
            C_pressure_inlet2=getFieldMean(P, "PRESSURE_PIPE21_85")
            C_pressure_outlet2=getFieldMean(P, "PRESSURE_PIPE22_1")

            # Mass flow rates
            C_massflow_inlet1=getFieldMean(P, "LIQFLOW_PIPE11_85")
            C_massflow_outlet1=getFieldMean(P, "LIQFLOW_PIPE12_1")
            C_massflow_inlet2=getFieldMean(P, "LIQFLOW_PIPE21_85")
            C_massflow_outlet2=getFieldMean(P, "LIQFLOW_PIPE22_1")

        # Enthalpies
```

Documentation of the Interface for Code Coupling : ICoCo

```
C_enthalpy_inlet1=getFieldMean(P,"LIQH_PIPE11_85")
C_enthalpy_outlet1=getFieldMean(P,"LIQH_PIPE12_1")
C_enthalpy_inlet2=getFieldMean(P,"LIQH_PIPE21_85")
C_enthalpy_outlet2=getFieldMean(P,"LIQH_PIPE22_1")
pass
```

```
C_pressure_inlet1 = mpi_bcast(C_pressure_inlet1,0)
C_pressure_outlet1 = mpi_bcast(C_pressure_outlet1,0)
C_pressure_inlet2 = mpi_bcast(C_pressure_inlet2,0)
C_pressure_outlet2 = mpi_bcast(C_pressure_outlet2,0)
C_massflow_inlet1 = mpi_bcast(C_massflow_inlet1,0)
C_massflow_outlet1 = mpi_bcast(C_massflow_outlet1,0)
C_massflow_inlet2 = mpi_bcast(C_massflow_inlet2,0)
C_massflow_outlet2 = mpi_bcast(C_massflow_outlet2,0)
C_enthalpy_inlet1 = mpi_bcast(C_enthalpy_inlet1,0)
C_enthalpy_outlet1 = mpi_bcast(C_enthalpy_outlet1,0)
C_enthalpy_inlet2 = mpi_bcast(C_enthalpy_inlet2,0)
C_enthalpy_outlet2 = mpi_bcast(C_enthalpy_outlet2,0)
```

```
# Compute Trio_U input values
# calculation of input temperature for Trio_U
temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref
```

```
# calculation of inlet and outlet velocity for Trio_U
u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio
```

```
# Modifications to ensure div(u)=0
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4
```

```
# Get output from Trio_U
T_pressure_inlet1 = 0.
T_pressure_outlet1 = 0.
T_pressure_inlet2 = 0.
T_pressure_outlet2 = 0.
T_temperature_inlet1 = 0.
T_temperature_inlet2 = 0.
T_temperature_outlet1 = 0.
T_temperature_outlet2 = 0.
if (progname=="Trio"):
    # Pressures
    T_pressure_inlet1=getFieldMean(P,"pressure_inlet1",comm_trio)*rho_Trio
    T_pressure_inlet2=getFieldMean(P,"pressure_inlet2",comm_trio)*rho_Trio
```

Documentation of the Interface for Code Coupling : ICoCo

```
T_pressure_outlet1=getFieldMean(P,"pressure_outlet1",comm_trio)*rho_Trio
```

```
T_pressure_outlet2=getFieldMean(P,"pressure_outlet2",comm_trio)*rho_Trio
```

```
# Temperatures
```

```
T_temperature_inlet1=getFieldMean(P,"temperature_inlet1",comm_trio)
```

```
T_temperature_inlet2=getFieldMean(P,"temperature_inlet2",comm_trio)
```

```
T_temperature_outlet1=getFieldMean(P,"temperature_outlet1",comm_trio)
```

```
T_temperature_outlet2=getFieldMean(P,"temperature_outlet2",comm_trio)
```

```
pass
```

```
T_pressure_inlet1 = mpi_bcast(T_pressure_inlet1, 1)
```

```
T_pressure_outlet1 = mpi_bcast(T_pressure_outlet1, 1)
```

```
T_pressure_inlet2 = mpi_bcast(T_pressure_inlet2, 1)
```

```
T_pressure_outlet2 = mpi_bcast(T_pressure_outlet2, 1)
```

```
T_temperature_inlet1 = mpi_bcast(T_temperature_inlet1, 1)
```

```
T_temperature_inlet2 = mpi_bcast(T_temperature_inlet2, 1)
```

```
T_temperature_outlet1 = mpi_bcast(T_temperature_outlet1, 1)
```

```
T_temperature_outlet2 = mpi_bcast(T_temperature_outlet2, 1)
```

```
# Give input to Cathare
```

```
# Trio_U -> Cathare retroaction on pressure
```

```
if (programe=="Cathare"):
```

```
    if (P.presentTime(>)>time_retro_P) :
```

```
        dP_T1 = T_pressure_inlet1-T_pressure_outlet1
```

```
        dP_T2 = T_pressure_inlet1-T_pressure_outlet2
```

```
        dP_T3 = T_pressure_inlet1-T_pressure_inlet2
```

```
        dP_C1 = C_pressure_inlet1-C_pressure_outlet1
```

```
        dP_C2 = C_pressure_inlet1-C_pressure_outlet2
```

```
        dP_C3 = C_pressure_inlet1-C_pressure_inlet2
```

```
        tau_coupl=0.1
```

```
        k_loc=0
```

```
        if (dt>tau_coupl):
```

```
            k_loc=1
```

```
        else:
```

```
            k_loc=dt/tau_coupl
```

```
# pressure retroaction on outlets
```

```
dP1 += (dP_C1-dP_T1)*k_loc
```

```
dP2 += (dP_C2-dP_T2)*k_loc
```

```
dP3 -= (dP_C3-dP_T3)*k_loc
```

```
setConstantField(P,"DPLEXT_PIPE12_1",dP1)
```

```
setConstantField(P,"DPLEXT_PIPE22_1",dP2)
```

```
setConstantField(P,"DPLEXT_PIPE21_86",dP3)
```

```
pass
```

Documentation of the Interface for Code Coupling : ICoCo

```
# Trio_U -> Cathare retroaction on temperature
if (P.presentTime(>time_retro_T) :
  T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
  T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
  T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
  T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)

# mesh numbers are growing from "left" to "right"
# overlapping domain on the "right"
# Loop 1
setConstantField(P,"OVHLEXT_PIPE11_85",-10)
setConstantField(P,"OVPLEXT_PIPE11_85",-10)
setConstantField(P,"ENTHEXT_PIPE11_85",T_enthalpie_inlet1)
# Loop 2
setConstantField(P,"OVHLEXT_PIPE21_85",-10)
setConstantField(P,"OVPLEXT_PIPE21_85",-10)
setConstantField(P,"ENTHEXT_PIPE21_85",T_enthalpie_inlet2)

# overlapping domain on the "left"
# Loop 1
setConstantField(P,"OVHLEXT_PIPE12_1",10)
setConstantField(P,"OVPLEXT_PIPE12_2",10)
setConstantField(P,"ENTHEXT_PIPE12_1",T_enthalpie_outlet1)
# Loop 2
setConstantField(P,"OVHLEXT_PIPE22_1",10)
setConstantField(P,"OVPLEXT_PIPE22_2",10)
setConstantField(P,"ENTHEXT_PIPE22_1",T_enthalpie_outlet2)
pass
pass

# Give input to Trio_U
# Velocities
if (programe=="Trio"):
  setConstantField(P,"u_inlet1",u_inlet1,1)
  setConstantField(P,"u_inlet2",u_inlet2,1)
  setConstantField(P,"u_outlet1",u_outlet1,1)
  setConstantField(P,"u_outlet2",u_outlet2,1)

# Temperatures
setConstantField(P,"temperature_inlet1",temperature_inlet1)
setConstantField(P,"temperature_inlet2",temperature_inlet2)
setConstantField(P,"temperature_outlet1",temperature_outlet1)
setConstantField(P,"temperature_outlet2",temperature_outlet2)
pass

# Solve next time step
ok =P.solveTimeStep()
ok = mpi_and(ok)
# Two ways : The resolution failed, try with a new dt or validate and go
# to next time step
if not(ok) :
  P.abortTimeStep()
else:
```

Documentation of the Interface for Code Coupling : ICoCo

```
P.validateTimeStep()
```

```
    pass                                # End loop on timestep size  
pass                                   # End loop on timesteps
```

```
P.terminate()
```

```
closeLib(handle)
```

5.9 Annex 6 : header and source file of the Salome ICoCoComponent

5.9.1 Header file : ICoCoComponent.hxx

```
#ifndef _ICoCoComponent_included_
#define _ICoCoComponent_included_

#include <vector>
#include <string>

namespace ParaMEDMEM {
    class MEDCouplingFieldDouble;
}

namespace ICoCo {
    class TrioField;
    class Problem;
}

class ICoCoComponent {
public :

    // interface Problem
    ICoCoComponent();
    virtual ~ICoCoComponent();
    // useful functions for Salome component
    virtual void openLib(const char * libname);
    virtual void closeLib();
    virtual double getFieldMean(const std::string& name);
    virtual void setConstantField(const std::string& name, double val, int
comp=0);

    // interface ICOCO functions
    virtual void setDataFile(const std::string& datafile);
    virtual void setMPIComm(void* mpicomm);
    virtual bool initialize();
    virtual void terminate();

    // interface UnsteadyProblem
    virtual double presentTime() const;
    virtual double computeTimeStep(bool& stop) const;
    virtual bool initTimeStep(double dt);
    virtual bool solveTimeStep();
    virtual void validateTimeStep();
    virtual bool isStationary() const;
    virtual void abortTimeStep();

    // interface IterativeUnsteadyProblem
    virtual bool iterateTimeStep(bool& converged);

    // interface Restorable
```

Documentation of the Interface for Code Coupling : ICoCo

```
virtual void save(int label, const std::string& method) const;
virtual void restore(int label, const std::string& method);
virtual void forget(int label, const std::string& method) const;

// interface FieldIO

virtual std::vector<std::string> getInputFieldsNames() const;
virtual void getInputFieldTemplate(const std::string& name,
ICoCo::TrioField& afield) const;
virtual void setInputField(const std::string& name, const ICoCo::TrioField&
afield);
virtual std::vector<std::string> getOutputFieldsNames() const;
virtual void getOutputField(const std::string& name, ICoCo::TrioField&
afield) const;

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;
private:
ICoCo::Problem * _theProblem;
void* _theHandle ;
};
#endif
```

5.9.2 Source File ICoCoComponent.cxx

Note that the functions `getFieldMean` and `setConstantField` don't have as first argument the ICoCo problem because there is only one problem for one ICoCoComponent and the problem is set in the `openLib` function (initialization of the private `_theProblem` attribute).

```
#ifndef ICoCoMEDCoupling
#define ICoCoMEDCoupling 1
#endif

#include <Problem.h>
#include "ICoCoTrioField.h"
#include <ICoCoComponent.hxx>

#include <sstream>
#include <string>
#include <exception>
#include <dlfcn.h>
#include <cstdio>
#include <cstdlib>
#include <iostream>

using namespace std;
using namespace ICoCo ;

ICoCoComponent::ICoCoComponent()
{
}

ICoCoComponent::~ICoCoComponent()
{
}

// OpenLib function
// Input parameter: the name of the library
// initialize _theProblem attribute
void ICoCoComponent::openLib(const char* libname)
{
    // open shared library
    void* handle;
    Problem *(*getProblem)();
    // open the code dynamic library
    handle =dlopen(libname, RTLD_LAZY | RTLD_LOCAL);
    if (!handle) {
        cerr << dlerror() << endl;
        throw 0;
    }
    // look at getProblem method in the code dynamic library
    getProblem=(Problem* (*)( ))dlsym(handle, "getProblem");
    if (!getProblem) {
        cout << dlerror() << endl;
        throw 0;
    }
}
```


Documentation of the Interface for Code Coupling : ICoCo

```
}
// instantiation of ICoCo Problem
_theProblem = (*getProblem)();
_theHandle = handle ;
}

// closeLib function
// No Input parameter
void ICoCoComponent::closeLib()
{
    if (dlclose(_theHandle)) {
        cout << dlerror() << endl;
        throw 0;
    }
}

// getFieldMean function
// input parameter : name of the field to get
// output parameter: the mean of the field (a double)
double ICoCoComponent::getFieldMean(const string& name)
{
    double mean=0.;
    TrioField f;

    // get the output field from the problem
    _theProblem->getOutputField(name,f);
    // only one component fields are treated
    if (f._nb_field_components!=1)
        throw 0;

    // Compute and return the mean value (all elements/nodes have the same
    weight)
    for (int loc=0;loc<f.nb_values();loc++)
        mean=mean+f._field[loc];
    return mean/f.nb_values();
}

// setConstantField function
// input parameters: 1 name of the field to set
//                  2 value to set
//                  3 component of the field to affect (1st one by default)
// output parameter: No output parameter

void ICoCoComponent::setConstantField(const string& name, double val, int
comp)
{
    TrioField f;
    // Get the right geometry and format for the field
    _theProblem->getInputFieldTemplate(name,f);

    // Allocate memory for the values (size=nb_elems*nb_comp)
    f.set_standalone();
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
// Fill the values
// Give a constant field to a Problem.
// One component of the field is specified, the others are 0.
int nb_elems=f._nb_elems;
int nb_comp=f._nb_field_components;
for (int i=0;i<nb_elems;i++)
  for (int j=0;j<nb_comp;j++)
    if (j!=comp)
      f._field[i*nb_comp+j]=0;
    else
      f._field[i*nb_comp+j]=val;

// Give the field to the problem
_theProblem->setInputField(name,f);
}

void ICoCoComponent::setDataFile(const string& datafile)
{
  _theProblem->setDataFile(datafile);
}

void ICoCoComponent::setMPIComm(void* mpicomm)
{
  _theProblem->setMPIComm(mpicomm);
}

bool ICoCoComponent::initialize()
{
  return _theProblem->initialize();
}

void ICoCoComponent::terminate()
{
  _theProblem->terminate();
  delete _theProblem;
}

double ICoCoComponent::presentTime() const
{
  return _theProblem->presentTime();
}

double ICoCoComponent::computeTimeStep(bool & stop) const
{
  return _theProblem->computeTimeStep(stop);
}

bool ICoCoComponent::initTimeStep(double dt)
{
  return _theProblem->initTimeStep(dt);
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
bool ICoCoComponent::solveTimeStep()
{
    return _theProblem->solveTimeStep();
}

void ICoCoComponent::validateTimeStep()
{
    _theProblem->validateTimeStep();
}

bool ICoCoComponent::isStationary() const
{
    return _theProblem->isStationary();
}

void ICoCoComponent::abortTimeStep()
{
    _theProblem->abortTimeStep();
}

bool ICoCoComponent::iterateTimeStep(bool& converged)
{
    return _theProblem->iterateTimeStep(converged);
}

void ICoCoComponent::save(int label, const std::string& method) const
{
    _theProblem->save(label,method);
}

void ICoCoComponent::restore(int label, const std::string& method)
{
    _theProblem->restore(label,method);
}

void ICoCoComponent::forget(int label, const std::string& method) const
{
    _theProblem->forget(label,method);
}

vector<string> ICoCoComponent::getInputFieldsNames() const
{
    return _theProblem->getInputFieldsNames();
}

void ICoCoComponent::getInputFieldTemplate(const std::string& name,
TrioField& afield) const
{
    _theProblem->getInputFieldTemplate(name,afield);
}
```

Documentation of the Interface for Code Coupling : ICoCo

```
void ICoCoComponent::setInputField(const std::string& name, const TrioField&
afield)
{
    _theProblem->setInputField(name, afield);
}
```

```
vector<string> ICoCoComponent::getOutputFieldsNames() const
{
    return _theProblem->getOutputFieldsNames();
}
```

```
void ICoCoComponent::getOutputField(const std::string& name, TrioField&
afield) const
{
    _theProblem->getOutputField(name, afield);
}
```

```
ParaMEMMEM::MEDCouplingFieldDouble*
ICoCoComponent::getInputMEDFieldTemplate(const string& name) const
{
    return _theProblem->getInputMEDFieldTemplate(name);
}
```

```
void ICoCoComponent::setInputMEDField(const string& name, const
ParaMEMMEM::MEDCouplingFieldDouble* afield)
{
    _theProblem->setInputMEDField(name, afield);
}
```

```
ParaMEMMEM::MEDCouplingFieldDouble* ICoCoComponent::getOutputMEDField(const
string& name) const
{
    return _theProblem->getOutputMEDField(name);
}
```

5.10 Annex 7: python script for a Salome use

```
import salome
import ICoCoComponent_ORB
salome.salome_init()

T = salome.lcc.FindOrLoadComponent("FactoryServer", "ICoCoComponent")
C = salome.lcc.FindOrLoadComponent("FactoryServer2", "ICoCoComponent")
# open shared libraries
T.openLib("./_Trio_UModule_opt.so")
C.openLib("./libcathare_gad.so")
# Initialize the problems
T.setDataFile("jdd_Trio.data")
T.initialize()
C.initialize()

# Coupling parameters
# Time for which Cathare runs alone. Allows to reach a Cathare steady state.
t_begin_trio=1000
# Time at which Trio_U -> Cathare retroaction on pressure begins
time_retro_P=1001
# Time at which Trio_U -> Cathare retroaction on temperature begins
time_retro_T=1003

# data to transform Cathare enthalpy into Trio_U temperature or vice versa
# H = href + Cp_Trio * ( T - Tref )
href = 1.15116e6
tref = 263.8+273.15 # Trio_U temperature are in K
Cp_Trio = 5332.43

# mass flowrate transformation
# In Cathare, pipe of diameter 0.6m (S=0.28274m2)
# In Trio_U, 2D pipe of section 0.6m
# We keep the same Q=rho*u*S between Cathare and Trio_U
# => u (Trio) = Q (Cathare) / S (Trio) / rho(Trio)
# !! u(Trio) != u(Cathare)
S_Trio = 0.6
rho_Trio = 739.206

# variables for pressure retroaction
# Must be remembered from timestep to timestep
dP1=0 # outlet1 - inlet1
dP2=0 # outlet2 - inlet1
dP3=0 # inlet2 - inlet1

# First time loop (Cathare alone)
epsilon=1e-10 # small time for double comparisons

while (1) : # Loop on timesteps
    present=C.presentTime()
```

Documentation of the Interface for Code Coupling : ICoCo

```
dt,stop =C.computeTimeStep()
if (stop or present>t_begin_trio-epsilon):
    break

# Modify dt in order to come to exactly t_begin_trio
if (present+dt>t_begin_trio):
    dt=t_begin_trio-present
elif (present+2*dt>t_begin_trio):
    dt=(t_begin_trio-present)/2

# Initialize the timestep
C.initTimeStep(dt)

# Perform the computation
ok=C.solveTimeStep()

# Either validate or abort it
if not(ok): # The resolution failed, try with a new dt
    C.abortTimeStep()
else:      # Validate and go to the next time step
    C.validateTimeStep()
pass      # End loop on timesteps

# Second time loop (both codes)
ok=True   # Is the time interval successfully solved?
stop=False

while not(stop) :           # Loop on timesteps
    ok=False
    while not(ok) :        # Loop on timestep size

        # Compute time step length
        dt_C, stop_C=C.computeTimeStep()
        dt_T, stop_T=T.computeTimeStep()
        dt=min(dt_C,dt_T)

        # And decide if we have to stop
        stop = stop_T or stop_C
        if (stop):
            break

        # prepare the new time step
        C.initTimeStep(dt)
        T.initTimeStep(dt)

        # Get output from Cathare
        # Pressures
        C_pressure_inlet1=C.getFieldMean("PRESSURE_PIPE11_85")
        C_pressure_outlet1=C.getFieldMean("PRESSURE_PIPE12_1")
        C_pressure_inlet2=C.getFieldMean("PRESSURE_PIPE21_85")
        C_pressure_outlet2=C.getFieldMean("PRESSURE_PIPE22_1")
```

Documentation of the Interface for Code Coupling : ICoCo

```

# Mass flow rates
C_massflow_inlet1=C.getFieldMean("LIQFLOW_PIPE11_85")
C_massflow_outlet1=C.getFieldMean("LIQFLOW_PIPE12_1")
C_massflow_inlet2=C.getFieldMean("LIQFLOW_PIPE21_85")
C_massflow_outlet2=C.getFieldMean("LIQFLOW_PIPE22_1")

# Enthalpies
C_enthalpy_inlet1=C.getFieldMean("LIQH_PIPE11_85")
C_enthalpy_outlet1=C.getFieldMean("LIQH_PIPE12_1")
C_enthalpy_inlet2=C.getFieldMean("LIQH_PIPE21_85")
C_enthalpy_outlet2=C.getFieldMean("LIQH_PIPE22_1")

# Compute Trio_U input values
#calculation of input temperature for Trio_U
temperature_inlet1=(C_enthalpy_inlet1-href)/Cp_Trio+tref
temperature_inlet2=(C_enthalpy_inlet2-href)/Cp_Trio+tref
temperature_outlet1=(C_enthalpy_outlet1-href)/Cp_Trio+tref
temperature_outlet2=(C_enthalpy_outlet2-href)/Cp_Trio+tref

# calculation of inlet and outlet velocity for Trio_U
u_inlet1 =C_massflow_inlet1 /S_Trio/rho_Trio
u_outlet1=C_massflow_outlet1/S_Trio/rho_Trio
u_inlet2 =C_massflow_inlet2 /S_Trio/rho_Trio
u_outlet2=C_massflow_outlet2/S_Trio/rho_Trio

# Modifications to ensure div(u)=0
# works because all areas are equal.
divergence=u_outlet1+u_outlet2-u_inlet1-u_inlet2
u_outlet1-=divergence/4
u_outlet2-=divergence/4
u_inlet1+=divergence/4
u_inlet2+=divergence/4

# Get output from Trio_U

# Pressures
T_pressure_inlet1=T.getFieldMean("pressure_inlet1")*rho_Trio
T_pressure_inlet2=T.getFieldMean("pressure_inlet2")*rho_Trio
T_pressure_outlet1=T.getFieldMean("pressure_outlet1")*rho_Trio
T_pressure_outlet2=T.getFieldMean("pressure_outlet2")*rho_Trio

# Temperatures
T_temperature_inlet1=T.getFieldMean("temperature_inlet1")
T_temperature_inlet2=T.getFieldMean("temperature_inlet2")
T_temperature_outlet1=T.getFieldMean("temperature_outlet1")
T_temperature_outlet2=T.getFieldMean("temperature_outlet2")

# Give input to Cathare
# Trio_U -> Cathare retroaction on pressure
if (C.presentTime(>time_retro_P) :
    dP_T1 = T_pressure_inlet1-T_pressure_outlet1
    dP_T2 = T_pressure_inlet1-T_pressure_outlet2
    dP_T3 = T_pressure_inlet1-T_pressure_inlet2

```

Documentation of the Interface for Code Coupling : ICoCo

```
dP_C1 = C_pressure_inlet1-C_pressure_outlet1
dP_C2 = C_pressure_inlet1-C_pressure_outlet2
dP_C3 = C_pressure_inlet1-C_pressure_inlet2

tau_coupl=0.1
k_loc=0
if (dt>tau_coupl):
    k_loc=1
else:
    k_loc=dt/tau_coupl

# pressure retroaction on outlets
dP1 += (dP_C1-dP_T1)*k_loc
dP2 += (dP_C2-dP_T2)*k_loc
dP3 -= (dP_C3-dP_T3)*k_loc
C.setConstantField("DPLEXT_PIPE12_1",dP1,0)
C.setConstantField("DPLEXT_PIPE22_1",dP2,0)
C.setConstantField("DPLEXT_PIPE21_86",dP3,0)
pass

# Trio_U -> Cathare retroaction on temperature
if (C.presentTime(>)>time_retro_T) :
    T_enthalpie_inlet1=href+Cp_Trio*(T_temperature_inlet1-tref)
    T_enthalpie_inlet2=href+Cp_Trio*(T_temperature_inlet2-tref)
    T_enthalpie_outlet1=href+Cp_Trio*(T_temperature_outlet1-tref)
    T_enthalpie_outlet2=href+Cp_Trio*(T_temperature_outlet2-tref)

# mesh numbers are growing from "left" to "right"
# overlapping domain on the "right"
# Loop 1
C.setConstantField("OVHLEXT_PIPE11_85",-10,0)
C.setConstantField("OVPLEXT_PIPE11_85",-10,0)
C.setConstantField("ENTHEXT_PIPE11_85",T_enthalpie_inlet1,0)
# Loop 2
C.setConstantField("OVHLEXT_PIPE21_85",-10,0)
C.setConstantField("OVPLEXT_PIPE21_85",-10,0)
C.setConstantField("ENTHEXT_PIPE21_85",T_enthalpie_inlet2,0)

# overlapping domain on the "left"
# Loop 1
C.setConstantField("OVHLEXT_PIPE12_1",10,0)
C.setConstantField("OVPLEXT_PIPE12_2",10,0)
C.setConstantField("ENTHEXT_PIPE12_1",T_enthalpie_outlet1,0)
# Loop 2
C.setConstantField("OVHLEXT_PIPE22_1",10,0)
C.setConstantField("OVPLEXT_PIPE22_2",10,0)
C.setConstantField("ENTHEXT_PIPE22_1",T_enthalpie_outlet2,0)
pass

# Give input to Trio_U
# Velocities
print 'initiatialisation variables Trio'
```


Documentation of the Interface for Code Coupling : ICoCo

```
T.setConstantField("u_inlet1",u_inlet1,1)
T.setConstantField("u_inlet2",u_inlet2,1)
T.setConstantField("u_outlet1",u_outlet1,1)
T.setConstantField("u_outlet2",u_outlet2,1)

# Temperatures
T.setConstantField("temperature_inlet1",temperature_inlet1,0)
T.setConstantField("temperature_inlet2",temperature_inlet2,0)
T.setConstantField("temperature_outlet1",temperature_outlet1,0)
T.setConstantField("temperature_outlet2",temperature_outlet2,0)

# Solve next time step
ok_T=T.solveTimeStep()
ok_C=C.solveTimeStep()
ok = ok_T and ok_C

# Two ways : The resolution failed, try with a new dt or validate and go
# to next time step
if not(ok) :
    T.abortTimeStep()
    C.abortTimeStep()
    print "Abort at time " + C.presentTime()
else:
    T.validateTimeStep()
    C.validateTimeStep()

    pass                    # End loop on timestep size
    pass                    # End loop on timesteps

T.terminate()
C.terminate()
```