Chapter 5

Coupling of Code_Aster

Code_Aster's source code is mostly written in Fortran and Python. There are two ways one could couple Code_Aster. The first one is to adapt the source code by introducing the calls to preCICE. The second option is to couple it at a higher level, adding the coupling calls to a command file that controls the flow of the simulation. A simulation in Code_Aster is controlled by a command file, which contains Code_Aster commands for creating the mesh and the model, for defining boundary and initial conditions, and calling the relevant solver. This command file can be extended with Python scripting[1] to add more advanced functionality.

Listing 5.1 shows the typical structure of a Code_Aster command file. First, the case is created by defining the mesh, the model, the materials, and the boundary and initial conditions. Then a list of time steps is created, and is provided together with the setup to the solver.

For the thesis, it was decided to couple Code_Aster through the command file, since it is the least invasive way, as it does not require modifying the source code at all. Therefore, the coupling of Code_Aster consisted in the implementation of a command file which contains coupling operations and calls to the thermal solver.

The Code_Aster adapter was implemented after testing the different boundary conditions with the CalculiX adapter, and it was decided that only the Robin coupling boundary condition would be implemented for Code_Aster, since it performed better than the other ones.

5.1 Description of the Adapter

The approach taken in this thesis is to split the command file into two command files:

- The **case definition** .comm file, which is in charge of the creation of the mesh, model, materials, initial and boundary conditions;
- The **adapter** .comm file, which wraps the solver call in a loop and triggers the coupling operations. This file is the main .comm file. The case definition .comm is read by the

[1]`http://www.code-aster.org/V2/doc/v12/en/man_u/u1/u1.03.02.pdf`

```
# Read the mesh
MESH = LIRE_MAILLAGE(...)

# Create the model
MODEL = AFFE_MODELE(..., MAILLAGE=MESH, ...)

# Define a material
MAT = DEFI_MATERIAU(...)

# Assign the material to the mesh
MATS = AFFE_MATERIAU(..., MAILLAGE=MESH,...)

# Create boundary conditions
BC = AFFE_CHAR_THER(..., MODELE=MODEL, ...)

# Create initial condition
INIT_T = CREA_CHAMP(...)

# Define a list of steps to solve
STEP = DEFI_LIST_REEL(...)

# Call the solver
TEMP = THER_LINEAIRE(..., MODELE=MODEL, ...)
```

Listing 5.1: Sample structure of a Code_Aster command file

adapter through the `INCLUDE` command, invoked at the beginning.

In practice, there is another .comm file, `config.comm`, used to configure the coupling. However, this is equivalent to the YAML configuration file used by the other two solvers, and does not have to do with the structure of the adapter.

Besides the adapter .comm file, a Python module was created, which is in charge of carrying out lower level accesses to the solver's data for the coupling. Therefore, the Code_Aster adapter consists actually of two files:

- `adapter.comm`
- `adapter.py`, which contains two classes:
  - `Adapter`, which handles the coupling
  - `Interface`, which handles the coupled surface meshes and their data

Listing 5.2 shows the general structure of the adapted Code_Aster. The solution process takes place in the call to `THER_LINEAIRE` for linear problems or `THER_NON_LINE` for non-linear problems. The procedure for steady-state problems is slightly different, but for simplicity, the checking of whether it is a linear or non-linear, steady-state or transient problem, has been omitted from the code snippet. Boundary conditions are updated before the call to the solver by calling `adapter.readCouplingData()`. Similarly, updated values are written to preCICE after solving, by calling `adapter.writeCouplingData()`. The time step size of the solver is always equal to the coupling time step size (returned by the `initialize` or `advance` methods of preCICE), which means that sub-cycling is not supported. This is, however, not a concern, because it is typically the fluid solver that needs sub-cycling. The `adapter.readCheckpoint()` and `adapter.writeCheckpoint()` functions do not really perform any checkpointing, but directly notify preCICE that the checkpoint reading/writing has been fulfilled.

5.2 Surface Mesh and Coupling Data

Data Locations

Thermal boundary conditions in Code_Aster are assigned through the comand `AFFE_CHAR_THER`[2]. A convective boundary condition was used for the Robin-Robin coupling. In Code_Aster, this corresponds to using a boundary of type `ECHANGE`.

The convective boundary condition is applied to the element face, and therefore the face center is used as the data location. However, for extracting the sink temperature and the heat transfer coefficient, it was more straightforward to do it on the element nodes. Therefore, for the coupling of Code_Aster, two meshes are used: one for the face centers and one for the nodes.

To extract the node coordinates from a Code_Aster MESH object:

```
MESH.sdj.COORDO.VALE.get()
```

Listing 5.3: Extracting the node positions in Code_Aster

To extract the face centers, connectivity information is used. First, the MESH object is transformed into Python object. The Python object allows to access the connectivity through the member variable `co`. The connectivity provides the three nodes that constitute a face. The coordinates of the nodes can be obtain from the member variable `cn` of mesh object. The face center is then computed as the average position between the three nodes.

```
mesh = MAIL_PY()
mesh.FromAster(MESH)
connectivity = [mesh.co[idx] for idx in mesh.gma[groupName]]
faceCenterCoordinates = np.array([np.array([mesh.cn[node] for node in
 ↪  elem]).mean(0) for elem in connectivity])
```

Listing 5.4: Extracting the face centers in Code_Aster

[2]http://www.code-aster.org/doc/v11/en/man_u/u4/u4.44.02.pdf

```
...

# Include the case definition .comm file
INCLUDE(UNITE=91)
...

# Reset time and set initial condition
k = 0
time = 0.0
ICOND = {'CHAM_NO': INIT_T}

while precice.isCouplingOngoing():

    adapter.writeCheckpoint()
    adapter.readCouplingData()
    ...

    # Call the linear thermal solver
    TEMP = THER_LINEAIRE(MODELE=MODEL,
        CHAM_MATER=MATS,
        EXCIT=LOADS,
        ETAT_INIT=ICOND,
        INCREMENT=_F(LIST_INST=STEP),
        PARM_THETA=1.0
    )
    T = CREA_CHAMP(RESULTAT=TEMP,
        NOM_CHAM='TEMP',
        TYPE_CHAM='NOEU_TEMP_R',
        OPERATION='EXTR',
        NUME_ORDRE=1
    )

    adapter.writeCouplingData(T)
    dt = adapter.advance()
    adapter.readCheckpoint()

    if adapter.isCouplingTimestepComplete():

        # Output if necessary ...

        # Set current solution as initial condition of next time step
        ...
        TEMP_CPY = COPIER(CONCEPT=TEMP)
        ICOND = {'EVOL_THER': TEMP_CPY}

        # Increment time
        time = time + dt
```

Listing 5.2: Code_Aster adapter command file

Boundary Conditions

In order to be able to change the values of the sink temperature and the heat transfer coefficient of the convective boundary condition, it is necessary to initially assign this type of boundary condition to the interface elements. Furthermore, they must be initialized with different values, otherwise Code_Aster will group them together, making it impossible to change the value corresponding to each individual element later. Additionally, `COEF_H` cannot be initialized with 0. Listing 5.5 shows how the initialization of the sink temperature (T_EXT) and the heat transfer coefficient (COEF_H) has been implemented, given a list of face elements. Notice how the parameters are initialized with different values by using the loop counter `j`. This initialization has to do with the allocation of memory and not to the physical initial values of the parameters. Initialization of the coupling data must be used (`initialize="yes"` in precice-config.xml), in order to override these values.

```
BCs = [
    {'MAILLE': faces[j], 'TEMP_EXT': j, 'COEF_H': j+1}
    for j in range(len(faces))
]
AFFE_CHAR_THER(MODELE=MODEL, ECHANGE=BCs)
```

Listing 5.5: Initialization of the convective boundary condition

This method works, but the implementation is not efficient, because Code_Aster is not efficient dealing with this element-by-element assignment[3], leading to a relatively long initialization time (e.g. compared to CalculiX). Future work may try to improve this implementation to make it more efficient. It is not certain whether this can be done at the command file level, or whether it must be done at the source code level.

During the coupling, the values of the `T_EXT` and `COEF_H` parameters have to be updated. This is shown in Listing 5.6, where `temp` and `hCoeff` are Python arrays.

The method changeJeveuxValues for changing the values of `T_EXT` and `COEF_H` has the following signature:

where `nbval` is the number of values, `indices` indicate the indices where the values must be changed, `real` is the array of values to assign to the real part, `imag` is the array of values to assign to the imaginary part (ignored in this case).

(Note that the hard-coded values 10 and 3 account for the spacing between the data values.)

Boundary Values

To compute the sink temperature to be sent to the coupling partner, temperatures at the interior of the solid are sampled. To obtain the points inside the solid, the nodes of the surface mesh are taken as starting points. The position of each node is then displaced by a

[3]A similar issue has been reported in `http://www.code-aster.org/forum2/viewtopic.php?id=18452`

```
L = AFFE_CHAR_THER(MODELE=MODEL, ECHANGE=BCs)
LOAD = {'CHARGE': L}
LOAD.sdj.CHTH.T_EXT.VALE.changeJeveuxValues(
    len(temp),
    tuple(np.array(range(len(temp))) * 10 + 1),
    tuple(temp),
    tuple(temp),
    1
)
LOAD.sdj.CHTH.COEFH.VALE.changeJeveuxValues(
    len(hCoeff),
    tuple(np.array(range(1, len(hCoeff)+1)) * 3 + 1),
    tuple(hCoeff),
    tuple(hCoeff),
    1
)
```

Listing 5.6: Update of coupling boundary values

```
changeJeveuxValues(self, nbval, indices, real, imag, num = 1)
```

Listing 5.7: changeJeveuxValues

prescribed distance δ in the direction opposite to the surface normal. During the coupling, the value of the temperature at the interior point is obtained by interpolation, using the shape functions of the element that contains the point. This interpolation is performed with the Code_Aster operator `PROJ_CHAMP`[4], using the method `COLLOCATION`.

```
PROJ_CHAMP(MAILLAGE_1=MESH, MAILLAGE_2=SHMESH, CHAM_GD=T,
 ↪  METHODE='COLLOCATION')
```

Listing 5.8: Computation of the sink temperature

To compute the heat transfer coefficient, the value of the thermal conductivity is divided by δ. The value of the conductivity is obtained as shown in Listing 5.9, through the method `RCVALE` of the material object `MAT`, which returns the conductivity as a function of the temperature `t`.

```
MAT.RCVALE("THER", nompar="TEMP", valpar=t, nomres="LAMBDA")[0][0]
```

Listing 5.9: Computation of the thermal conductivity

[4]`http://www.code-aster.org/doc/v12/en/man_u/u4/u4.72.05.pdf`

Notes:

- The value of δ has been fixed to 1e-5. Future development may try to adjust this value based on the size of the mesh or elements.
- The interface is associated to only one material model (e.g. one MAT object). It cannot handle the case where more than one material is assigned to the interface, mainly because no way was found to obtain the material model that was assigned to a particular face element (therefore it is assumed that all faces have the same material model).

5.3 Parallelization

In Code_Aster, it is possible to use both shared memory (OpenMP) and distributed memory (MPI) parallelization. This can be chosen at run time, through the configuration of the solver[5]. Additionally, the keyword `PARTITION` in the command `AFFE_MODELE`[6] allows to define how the elements will be distributed in the parallel phases of Code_Aster.

In the adapter implemented in this thesis, the Code_Aster participant only supports shared memory parallelization. From preCICE's point of view it would be just a serial participant. Due to time limitations, it was not possible to find a straightforward way to do the domain decomposition and have each process access only its own part of the mesh. This is the first step required to support parallelism in the coupling adapter. This may be addressed in future work.

5.4 Steady-State Simulation

In order to run a steady-state simulation in the coupled Code_Aster, this must be specified in the input config.comm file (or in the config.yml.org file if the auto-config.py utility is used). The adapter will check whether `steady-state` is set to true or false in the configuration file, and will call the Code_Aster solvers accordingly.

The configuration files are explained in more detail in Chapter 6.

5.5 Non-Linear Solver

Similar to using the steady-state coupling, the use of the non-linear solver is activated from the configuration file (config.comm or config.yml.org).

The configuration files are explained in more detail in Chapter 6.

[5]`http://www.code-aster.org/doc/v12/en/man_u/u4/u4.50.01.pdf`
[6]`http://www.code-aster.org/doc/v12/en/man_u/u4/u4.41.01.pdf`

Chapter 6

Case Setup

6.1 Structure and Configuration

During the setup of a CHT simulation, each participant is prepared independently. However, one must make sure that the correct boundary conditions are used in each solver, so that later the updated boundary values can be applied during the coupling.

Besides preparing the case folder for each participant, two configuration files are needed:

- `precice-config.xml`: Used by the preCICE library for configurint the meshes, the data and the coupling schemes.
- `config.yml`: Contains additional information required by the adapter which is not in the precice-config.xml file.

Therefore, the structure of a preCICE-coupled CHT case looks like:

```
cht-case
    precice-config.xml
    config.yml
    run.sh
    fluid
        0
        constant
        system
    solid
        all.mesh
        solid.inp
        interface.nam
        interface.sur
        interface.flm
        ...
```

6.2 Automatic Generation of Configuration Files

A Python utility for automatically generating the above-mentioned configuration files from a base configuration file was developed. This utility, `auto-config.py`, reads as an input another YAML file with minimal information. The rest of the information for `precice-config.xml` and `config.yml` is generated by the utility, which tries to set good default values.

The usage of the utility:

```
python auto-config.py
    [--input-file default:config.yml.org]
    [--output-xml-file default:precice-config.xml]
    [--output-yml-file default:config.yml]
    [--output-comm-file default:config.comm]
    [--output-sh-file default:run.sh]
```

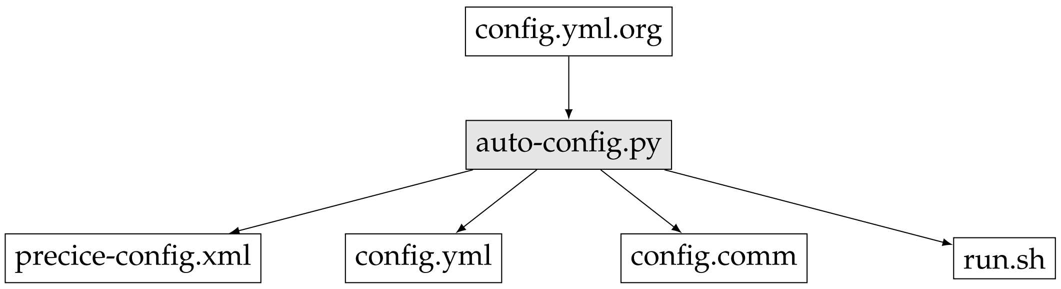

Figure 6.1: Input and output files of the auto-config utility

Note: The reason why there is an input YAML and an output YAML is that additional information is attached to the output YAML file. The auto-config utility always generates a YAML file for Robin-Robin coupling and makes some assumptions regarding the naming of the meshes and coupling data. The config.yml that is accepted by the adapter is actually more generic.

6.2.1 Input File

A sample input config.yml.org file is provided below:

Listing 6.1 Sample config.yml.org file for the heat exchanger case (OpenFOAM-CalculiX)

```
simulation:

    time-step: 1
    max-time: 500
    steady-state: True
    force-explicit: False
    force-parallel: False
    max-coupling-iterations: 30
    output-frequency: 10
    non-linear: False


participants:

    Inner-Fluid:
        solver: OpenFOAM
        interfaces:
            - {name: Inner-Fluid-to-Solid, patches: [interface]}
        domain-decomposed: True
        nprocs: 8
        directory: inner-fluid

    Outer-Fluid:
        solver: OpenFOAM
        interfaces:
            - {name: Outer-Fluid-to-Solid, patches: [interface]}
        domain-decomposed: True
        nprocs: 8
        directory: outer-fluid

    Solid:
        solver: CalculiX
        interfaces:
            - {name: Solid-to-Inner-Fluid, patch: inner-interface}
            - {name: Solid-to-Outer-Fluid, patch: outer-interface}
        domain-decomposed: False
        directory: solid


couplings:
    - [Inner-Fluid-to-Solid, Solid-to-Inner-Fluid]
    - [Outer-Fluid-to-Solid, Solid-to-Outer-Fluid]
```

General parameters that can be configured:

- time-step
- max-time
- steady-state
- output-frequency: Currently only used by the Code_Aster participant. This could be read by another utility to update the controlDict or the *.inp file for OpenFOAM and CalculiX, respectively.

Coupling parameters that can be configured:

- max-coupling-iterations: specifies the maximum number of coupling iterations when using implicit coupling
- force-explicit: implicit coupling will be used *whenever* possible, unless force-explicit is

set to True

- force-parallel: serial coupling will be used *whenever* possible, unless force-parallel is set to True

Fixed parameters (currently specified inside the Python code):

- IQN-ILS as convergence acceleration scheme for implicit coupling (its parameters are also fixed)
- Nearest-neighbor as data mapping scheme

Participant parameters:

- domain-decomposed: whether the participant is run in parallel. This is currently only supported by OpenFOAM. Furthermore, the case must already be decomposed by running decomposePar. This could be automated in the future by adding a call to decomposePar in the output run.sh script. (However, if the case is already decomposed, should we run decomposePar with the -force parameter? This would erase existing processor directories.)
- directory: case directory of the participant
- non-linear (only for Code_Aster): determines whether to use THER_LINEAIRE or THER_NON_LINEAI
- material-id (only for Code_Aster): must be specified for each interface. The ID corresponds to the index used in the definition of the materials in the .comm file.
- interfaces: a list of the interfaces must be provided. For each interface, a name must be provided (can be arbitrarily chosen). In the case of OpenFOAM, the interface may consist of several boundary patches. These must be specified as a list. In the case of CalculiX and Code_Aster, only one patch can be defined (a "surface" in CalculiX's terminology or a "GROUP_MA" (elements group) in Code_Aster)

Couplings:

- Pairs of coupled *interfaces* must be specified

6.2.2 Output Files

- precice-config.xml: preCICE configuration file
- config.yml: adapter configuration file used by OpenFOAM and CalculiX participants
- config.comm: adapter configuration file used by Code_Aster participants
- run.sh: script to run the coupled simulation (executes all involved participants with the appropriate command line arguments and environment variables, and directs the output into a log file)

Below, an example of the output config.yml file is shown. The color coding is as follows:

- Gray: Only necessary in the config.yml.org file that is read by the utility. This information is mainly to generate the precice-config.xml file. This information is not used by the adapters, and therefore could be omitted from the config.yml if it were created manually.

- Red: Information that is generated by the utility. This information is required by the adapters and therefore must be included in the YAML file that is read by them.

Listing 6.2 Sample output config.yml file of the auto-config.py utility

```
base-path: /home/lcheung/Thesis/precice-cases/heat-exchanger-
   par

precice-config-file: precice-config.xml

simulation: {force-explicit: false, force-parallel: true, max-
   coupling-iterations: 30,
  max-time: 2, non-linear: false, output-frequency: 10, steady-
     state: true, time-step: 1}

participants:

  Inner-Fluid:
    solver: OpenFOAM
    directory: inner-fluid
    domain-decomposed: true
    nprocs: 2
    interfaces:
    - name: Inner-Fluid-to-Solid
      mesh: Inner-Fluid-to-Solid
      patches: [interface]
      read-data: [Sink-Temperature-Solid, Heat-Transfer-
         Coefficient-Solid]
      write-data: [Sink-Temperature-Inner-Fluid, Heat-Transfer-
         Coefficient-Inner-Fluid]

  Outer-Fluid:
    solver: OpenFOAM
    directory: outer-fluid
    domain-decomposed: true
    nprocs: 2
    interfaces:
    - name: Outer-Fluid-to-Solid
      mesh: Outer-Fluid-to-Solid
      patches: [interface]
      read-data: [Sink-Temperature-Solid, Heat-Transfer-
         Coefficient-Solid]
      write-data: [Sink-Temperature-Outer-Fluid, Heat-Transfer-
         Coefficient-Outer-Fluid]
  Solid:
    solver: CalculiX
    directory: solid
    domain-decomposed: false
    interfaces:
    - name: Solid-to-Inner-Fluid
      mesh: Solid-to-Inner-Fluid
      patch: inner-interface
      read-data: [Sink-Temperature-Inner-Fluid, Heat-Transfer-
         Coefficient-Inner-Fluid]
      write-data: [Sink-Temperature-Solid, Heat-Transfer-
         Coefficient-Solid]
    - name: Solid-to-Outer-Fluid
      mesh: Solid-to-Outer-Fluid
      patch: outer-interface
      read-data: [Sink-Temperature-Outer-Fluid, Heat-Transfer-
         Coefficient-Outer-Fluid]
      write-data: [Sink-Temperature-Solid, Heat-Transfer-
         Coefficient-Solid]

couplings:
- [Inner-Fluid-to-Solid, Solid-to-Inner-Fluid]
- [Outer-Fluid-to-Solid, Solid-to-Outer-Fluid]
```

As already mentioned, the auto-config.py utility always generates Robin-Robin boundary conditions. However, the config.yml could also be configured to use Dirichlet-Neumann coupling, if `Temperature` and `Heat-Flux` are used as read/write data. Nevertheless, this also requires changes to the precice-config.xml file: the data names must be changed, and possibly other meshes have to be defined (e.g. a CalculiX participant uses a nodes-mesh and a faces-mesh for DN coupling). For a sample configuration for Dirichlet-Neumann coupling, the reader is referred to the flat-plate validation case, which has been tested for different coupling boundary conditions.

6.3 Relationship Between the Solver Boundaries and the preCICE Interfaces

- **OpenFOAM**: an interface can consist of multiple boundary patches. The grouping of several patches to form an interface is specified in the config.yml (or config.yml.org) file. For example, if the top, bottom, left and right patches make up the interface, we specify in the config.yml file:

```
interfaces:
    - {name: Fluid-to-Solid, patches: [top, bottom, left, right
       ]}
```

- **CalculiX**: a node set and a face set must be created for the interface nodes and faces respectively. If CalculiX's graphical interface cgx is used to setup the case, the sets are exported as follows:

  ```
  send interface abq nam
  send interface abq sur
  ```

  These commands create a node set with the name prepended with an N (i.e. Ninterface) and a surface set with the name prepended with an S (i.e. Sinterface), respectively. *Important*: If other tool is used to generate the sets, the set names must still have the N or S prefixes!

  In the config.yml file, only one set (patch) is specified, without the prefix:

  ```
  interfaces:
      - {name: Fluid-to-Solid, patch: interface}
  ```

- **Code_Aster**: similar to CalculiX, a group must be created for the face elements belonging to the interface (GROUP_MA). The nodes group is created automatically by the adapter. Same as in CalculiX, only one group (or patch) is specified in the config.comm (or config.yml.org) file.

6.4 How Many Participants to Use

In every simulation, we will have at least two regions (typically a fluid and a solid region). There are some cases where more than two physical regions need to be simulated. We have different possibilities on how to assign the regions to different participants:

- How many fluid participants to use:
  - One participant can be used for all fluid regions
    - If they all have the same properties (e.g. thermophysical properties)
    - If there are no fluid-fluid interfaces
  - Multiple participants must be used
    - If the regions have different properties (e.g. thermophysical properties)
    - If there are fluid-fluid interfaces
- How many solid participants to use:
  - One participant may be used; if there are solid-solid interfaces, they can be treated as contacts
  - Multiple participants may be used; the interfaces are coupled with preCICE (only perfect contact can be modeled in this case). Note: at the moment, solid-solid coupling via preCICE may not be very stable.

Another way to summarize these guidelines, is in terms of the interfaces:

- Fluid-solid interfaces: must be coupled through preCICE
- Fluid-fluid interfaces*: must be coupled through preCICE
- Solid-solid interfaces: can be coupled either through preCICE or by using contacts (see Appendix H)

*The current implementation couples the buoyantPimpleFoam solver, which does not support multiple regions. In future developments, one might couple chtMultiRegionFoam in order to be able to simulate fluid-fluid interfaces within one solver instance, instead of using an external coupling. This would require changes in the adapter.

6.5 Setting up an OpenFOAM Participant

Boundary Conditions

The boundary condition to be used for the coupling must be defined on the interface patch(es) of the temperature field in the T file (e.g. `0/T`). It is important to setup the correct type, otherwise the solver will crash when the adapter tries to update the values with the coupling data. Regarding the values to be used, one would typically set it the same way as setting initial conditions. It is also recommended to do an initial exchange of the boundary values, by setting `initialize="yes"` in the `exchange` tag(s) in the preCICE configuration file.

For a Dirichlet coupling boundary condition, the interface must have a boundary of type `fixedValue`.

Listing 6.3 Dirichlet boundary condition

```
interface
{
    type fixedValue;
    value uniform 310;
}
```

For a Neumann coupling boundary condition, the interface must have a boundary of type `fixedGradient`.

Listing 6.4 Neumann boundary condition

```
interface
{
    type fixedGradient;
    value uniform 0;
}
```

For a Robin coupling boundary condition, the interface must have a boundary of type `mixed`. Please note that for steady-state coupling, only Robin coupling boundary conditions are supported.

Listing 6.5 Mixed boundary condition

```
interface
{
    type mixed;
    value uniform 293;
    refValue uniform 310;
    refGradient 0; // must be 0!
    valueFraction 0.5;
}
```

As already mentioned, an interface may consist of multiple patches. Each coupled patch must be configured as explained above.

6.6 Setting up a CalculiX Participant

To setup a CalculiX participant, besides creating the case as usual, for each interface it is necessary to create the following files, and include them in the case .inp file

- A node set (e.g. interface.nam)
- A surface set (e.g. interface.sur)

Boundary Conditions

- If Dirichlet boundary condition is used, the interface node set must have the temperature boundary condition defined in the .inp file:

  ```
  *BOUNDARY
  Ninterface,11,11,300
  ```

- If Neumann boundary condition is used, the interface elements must have a distributed flux (DFLUX) boundary condition, normally defined in a .dfl file, which has to be included in the .inp file:

  ```
  *DFLUX
  *INCLUDE, INPUT=solid/interface.dfl
  ```

  A sample of the .dfl file is provided bellow. Each line contains the element ID, the face ID and the value of the heat flux (0 in this case).

  ```
  ** DFlux based on interface
  ** DFlux based on interface
  7118, S1, 0.000000e+00
  2781, S4, 0.000000e+00
  6014, S1, 0.000000e+00
  ```

```
6178, S1, 0.000000e+00
3818, S2, 0.000000e+00
...
```

- If Robin boundary condition is used, the interface elements must have a film (FILM) boundary condition, normally defined in a .flm file:

  ```
  *FILM
  *INCLUDE, INPUT=solid/interface.flm
  ```

  The structure of the .flm file is similar to that of the .dfl file, but in this case it would define the sink temperature and the heat transfer coefficient, instead of the heat flux.

6.7 Setting up a Code_Aster Participant

The files required to setup a Code_Aster participant are listed below. Note that input and output files in Code_Aster are associated with a logical unit number.

- The adapter.comm file, which must be read from UNIT 1
- The mesh file with extension .mmed, which must be read from UNIT 20
- The .comm file which contains the normal case setup, where the model, the material, the initial and boundary conditions, etc. are defined. It is like a normal Code_Aster, but without calling the solver. This must be read from UNIT 91. This file must define the following Code_Aster named concepts (variables):
  - MESH
  - MODEL
  - MAT[] (list of materials)
  - BC[] (list of boundary conditions)
  - INIT_T (initial condition for the temperature)
- Additionally, a third .comm file must be read, the config.comm file, which contains information required by the adapter. This is equivalent to the config.yml file used by the OpenFOAM and the CalculiX adapter. (There was an error loading the Python YAML library from the adapter.comm file, and therefore instead of a YAML file, it reads an equivalent .comm file). This file must be read from UNIT 90.

Notice that the main .comm file is the adapter. It contains the timestepping / coupling loop and makes calls to the solver. The adapter can be reused across different simulations. Inside the adapter, there are commands to include the other two .comm files, which are case specific and have to be created for each simulation.

Listing 6.6 shows how these files and their logical unit are specified in the .export file. The name of the last two .comm files is not important, as long as the correct UNIT is specified (the number at the end of each line in the snippet of the .export file).

Listing 6.6 Specification of input files in the Code_Aster .export file

```
...
F comm /path/to/adapter/adapter.comm D  1
F comm /path/to/case/config.comm D  90
F mmed /path/to/case/participant-dir/mesh.mmed D  20
F comm /path/to/case/participant-dir/case.comm D  91
...
```

The config.comm file is generated by the utility for automatic generation of the configuration files.

The adapter will always assume that Robin boundary condition is used for the coupling, therefore it is not necessary to specify any boundary condition for the interfaces. However, in the creation of the mesh, groups of faces must be created for the interfaces (i.e. a name must be given to the set of faces that belong to an interface). For example, if Salome-Meca is used for the mesh creation, one would typically use the "Create Groups from Geometry" option. Then, to tell the adapter that this face group has to be coupled, it must be specified in the config.comm file (or in the config.yml.org file if the auto-config.py utility is used).

6.8 Execution

To run one of the coupled solvers, it is necessary to provide:

- Name of the participant (as a command line argument for OpenFOAM and CalculiX, as an environment variable for Code_Aster)
- Name of the YAML configuration file (if not provided, it is assumed to be `config.yml`). (The name of the preCICE XML configuration file is provided in the YAML file.)

6.8.1 OpenFOAM

From the CHT case root directory, we execute the OpenFOAM solver with:

```
[foam solver] -case [case directory] -precice-participant [
   participant name]
```

For example:

```
buoyantPimpleFoam_preCICE -case fluid -precice-participant Fluid
buoyantSimpleFoam_preCICE -case fluid -precice-participant Fluid
```

A different YAML configuration file might be provided with `-config-file [my-config.yml]`.

6.8.2 CalculiX

From the CHT case root directory, we execute the CalculiX participant with:

```
ccx_preCICE -i [.inp file without the extension] -precice-participant
    [participant name]
```

For example:

```
ccx_preCICE -i solid/solid -precice-participant Solid
```

A different YAML configuration file might be provided with `-config-file [my-config.yml]`.

6.8.3 Code_Aster

To run a Code_Aster participant, first, the environment variable PRECICE_PARTICIPANT must be set with the name of the Code_Aster participant. Then, the simulation is run with `as_run`, and providing the name of the .export file.

```
export PRECICE_PARTICIPANT=Solid
as_run --run solid/solid.export
```

(There was no straightforward way to make the adapter read additional command line arguments, as in the case of OpenFOAM and CalculiX. Therefore, an environment variable is used.)

When coupling a Code_Aster participant, it is important to explicitly set the `exchange-directory` in the precice-config.xml file, because the Code_Aster case is actually copied and executed in a temporary directory. This is automatically taken care of by the auto-config.py utility.

6.8.4 Run Script

If the `auto_config.py` utility is used, the above explained commands are automatically written into a .sh script. To execute the coupled simulation:

```
./run.sh
```

The output of each solver is redirected to a log file (e.g. participant.log). The progress of the coupled simulation can be monitored by using:

```
tail -f [participantName].log
```