A. Details on Integrating the Coupling Adapter into SU^2

Changes Concerning SU^2 Configuration

The necessary code changes in SU^2 for using the four new configuration options described in Section 5.1 need to be applied to the class *CConfig*. However, the class is defined by a header (*config_structure.hpp*), an inline (*config_structure.inl*) and a source file (*config_structure.cpp*). All of them have to be adapted slightly. In *config_structure.hpp* the member variables of *CConfig* are declared. In order to store the values of the above mentioned configuration options, first of all, four new private variables need to be introduced. Moreover, public getter functions are needed to access these variables. The necessary changes are shown in the following:

```
class CConfig {
private:
  ...
  bool precice_usage;
  bool precice_verbosityLevel_high;
  string preciceConfigFileName;
  string preciceWetSurfaceMarkerName;
  ...
public:
  ...
  bool GetpreCICE_Usage(void);
  bool GetpreCICE_VerbosityLevel_High(void);
  string GetpreCICE_ConfigFileName(void);
  string GetpreCICE_WetSurfaceMarkerName(void);
  ...
};
```

Listing A.1: Code changes in *config_structure.hpp*.

The four simple getter functions declared above are implemented in the file *config_structure.inl*:

```
...
inline bool CConfig::GetpreCICE_Usage(void){return precice_usage;}

inline bool CConfig::GetpreCICE_VerbosityLevel_High(void){return
precice_verbosityLevel_high;}

inline string CConfig::GetpreCICE_ConfigFileName(void){return
preciceConfigFileName;}

inline string CConfig::GetpreCICE_WetSurfaceMarkerName(void){return
preciceWetSurfaceMarkerName;}
...
```

Listing A.2: Code changes in *config_structure.inl*.

Eventually, the configuration file interpreter needs to be informed about the new options. In *config_structure.cpp* an assignment for each of the configuration file options to the member variables of *CConfig* is defined as well as the default values. This must be done in *CConfig::SetConfig_Options* for the four new options as follows:

```
...
void CConfig::SetConfig_Options(...) {
  ...
  addBoolOption("PRECICE_USAGE", precice_usage, false);

  addBoolOption("PRECICE_VERBOSITYLEVEL_HIGH",
  precice_verbosityLevel_high, false);

  addStringOption("PRECICE_CONFIG_FILENAME", preciceConfigFileName,
  string("precice.xml"));

  addStringOption("PRECICE_WETSURFACE_MARKER_NAME",
  preciceWetSurfaceMarkerName, string("wetSurface"));
  ...
}
```

Listing A.3: Code changes in *config_structure.cpp*.

The new mesh movement option PRECICE_MOVEMENT in SU^2 as introduced in Section 5.1 needs to be added to the configuration procedure. In contrast to the other options stated above, no changes to the *CConfig* class are needed since the option name GRID_MOVEMENT_KIND already exists. However, the new value PRECICE_MOVEMENT must be introduced. Therefore, an adaption in *option_structure.hpp* is necessary. Whenever multiple values[1] for an option are possible in the SU^2 configuration file, internally each value is assigned with an identifying number. After parsing the configuration file, such option values are mapped to their respective identifiers. These are further used in the source code of SU^2. The *option_structure.hpp* file is included in *config_structure.hpp*. This way, *CConfig* can use such value mappings, although they are not directly defined in its own source code. The following changes simply extend the mapping of already existing values for GRID_MOVEMENT_KIND by another entry:

```
...
enum ENUM_GRIDMOVEMENT {
NO_MOVEMENT = 0,
...
RIGID_MOTION = 2,
...
ROTATING_FRAME = 8,
...
PRECICE_MOVEMENT = 13
};

static const map<string, ENUM_GRIDMOVEMENT> GridMovement_Map =
CCreateMap<string, ENUM_GRIDMOVEMENT>
("NONE", NO_MOVEMENT)
...
("RIGID_MOTION", RIGID_MOTION)
...
("ROTATING_FRAME", ROTATING_FRAME)
...
("PRECICE_MOVEMENT", PRECICE_MOVEMENT);
```

Listing A.4: Code changes in *option_structure.hpp*. Only the entries corresponding to PRECICE_MOVEMENT need to be added. All other options are shown for illustrative reasons. 13 is assigned to the new option value because of consecutive numbering. The class *CCreateMap* defines how the mapping is done technically. It is not described here, as this detail is not necessary.

[1]Note that "value" can correspond to strings such as PRECICE_MOVEMENT, see Equation 4.10.

The new mesh movement sequence as mentioned in Section 5.1 is defined as follows: A switch-case-statement in *iteration_structure.cpp* determines, which movement procedure to use based on the configuration information. The list of possible cases is now extended by PRECICE_MOVEMENT as follows:

```
void SetGrid_Movement(CGeometry **geometry_container, ...,
CVolumetricMovement *grid_movement, CConfig *config_container, ...,
unsigned long ExtIter) {
  ...
  unsigned short Kind_Grid_Movement =
  config_container->GetKind_GridMovement(...);
  ...
  switch (Kind_Grid_Movement) {
  ...
  case ROTATING_FRAME:
    ...
  case RIGID_MOTION:
    ...
  case PRECICE_MOVEMENT:
    grid_movement->SetVolume_Deformation(geometry_container[MESH_0],
    config_container, true);
    geometry_container[MESH_0]->SetGridVelocity(config_container,
    ExtIter);
    grid_movement->UpdateMultiGrid(geometry_container, config_container);
    break;

  case NO_MOVEMENT:
    ...
}
```

Listing A.5: Code changes in *iteration_structure.cpp*. Only the entries corresponding to PRECICE_MOVEMENT need to be added. All other options are shown for illustrative reasons.

Adaption of the Main Solver Routine of SU^2

First of all, the header file of the adapter, *precice.hpp*, needs to be included so that the adapter can be instantiated within the code of the solver:

```
#include "../include/precice.hpp"
```

Listing A.6: Including the header file of the adapter, *precice.hpp* in *SU2_CFD.cpp*.

For positioning newly added lines of code in *SU2_CFD.cpp* I refer to Algorithm 2. The initialization of the boolean flag, which determines whether preCICE should be used and the conditional startup of the coupling via the adapter is achieved by the following lines, which have to be inserted right before the main solver while-loop of SU^2 (i.e. between lines 8 and 9 of Algorithm 2):

```
bool precice_usage = config_container[ZONE_0]->GetpreCICE_Usage();
Precice *precice;
double *max_precice_dt, *dt;
if (precice_usage) {
  precice = new Precice(rank, size, geometry_container,
  solver_container, config_container, grid_movement);
  dt = new double(config_container[ZONE_0]->GetDelta_UnstTimeND());
  precice->configure(config_container[ZONE_0]->GetpreCICE_ConfigFileName());
  max_precice_dt = new double(precice->initialize());
}
```

Listing A.7: Insertion of the declarations and memory allocation of coupling-related variables, as well as startup of the coupling procedure.

The flag *precice_usage* is set via the newly added getter function *GetpreCICE_Usage()* from the *config_container*, which stores all configuration information. Upon initialization of the adapter object *precice*, MPI rank and size as well as the containers holding geometry, solver, configuration and grid movement information are passed to it. For configuration of preCICE, name and location of its configuration file need to be forwarded to the adapter. This is done by calling the new getter function *GetpreCICE_ConfigFileName()*. The variable *dt* refers to the current physical time step size in SU^2 and is needed for timing issues, as is *max_precice_dt*.

Next, the condition of the main solver while-loop (starting at line 9 of Algorithm 2) must be modified such that preCICE is able to shut down SU^2 if a simulation should be ended. This is the case if *isCouplingOngoing()* evaluates to false:

```
while ((ExtIter < config_container[ZONE_0]->GetnExtIter() &&
precice_usage && precice->isCouplingOngoing()) || (ExtIter <
config_container[ZONE_0]->GetnExtIter() && !precice_usage)) {
  ...
}
```

Listing A.8: Modification of main solver while-loop condition, which allows to shut down SU^2 via preCICE.

SU^2 cannot differentiate, whether a solver iteration is a new time step or a coupling subiteration. Therefore, preCICE is solely in charge of this decision and signalizes it via a flag. The following lines must be added right after the beginning of the while-loop body (between lines 9 and 10, Algorithm 2):

```
if(precice_usage && precice->isActionRequired(precice->getCowic())){
  precice->saveOldState(&StopCalc, dt);
}
```

Listing A.9: Saving the current solver state for checkpointing of implicit solver strategies in preCICE.

preCICE signals the necessity of saving the solver state via the *isActionRequired()* function. Its argument specifies that the required action relates to <u>w</u>riting an <u>i</u>teration <u>c</u>heckpoint (*getCo<u>wic</u>()*).

The following lines need to be inserted right after the code of Listing A.9 in order to allow preCICE to enforce time steps[2] smaller than the current value in SU^2:

```
if(precice_usage){
  dt = min(max_precice_dt, dt);
  config_container[ZONE_0]->SetDelta_UnstTimeND(*dt);
}
```

Listing A.10: Determining the minimal time step size for the next solver run.

preCICE is triggered, once a new fluid solution is computed:

```
if(precice_usage){
  *max_precice_dt = precice->advance(*dt);
}
```

Listing A.11: Advancing preCICE after a solver run of SU^2.

This code excerpt must be added between lines 14 and 15 of Algorithm 2, i.e. after convergence in SU^2 is checked, but before output files are written. *advance()* uses the current solver time step size as input and returns the new maximum limit for the next time instance. This is determined by comparing time step sizes of the coupled solvers up to the next instance when coupling data needs to be exchanged (compare Figure 4.2).

If a subiteration has not converged in preCICE, SU^2 is reset to the last iteration checkpoint. Thus, the following code is the counterpart of the *saveOldState()* function of Listing A.9 and must be inserted right after the *advance()* step:

[2] *dt* refers to the black and red arrows of Figure 4.2, while the green arrow corresponds to *max_precice_dt*.

```
if(precice_usage && precice->isActionRequired(precice->getCoric())){
  ExtIter--;
  precice->reloadOldState(&StopCalc, dt);
}
else if (solutionNeedsToBeOutput){
  writeOutputFiles();
}
```

Listing A.12: Reloading the old SU^2 solver state if another subiteration is necessary. Note that *solutionNeedsToBeOutput* and *writeOutputFiles()* are simplified pseudo-code expressions.

By analogy with *saveOldState()*, *reloadOldState()* is initiated by preCICE as it signalizes if an action is required. In contrast, this time the task does not relate to writing but rather reading an iteration checkpoint (*getCoric()*).

The following code is executed at the end of a coupled simulation and, thus, must be added between lines 22 and 23 of Algorithm 2:

```
if(precice_usage){
  precice->finalize();
  delete [] precice;
}
```

Listing A.13: Shutting down the coupling between SU^2 and preCICE and deallocating memory.

Algorithm 4 sums up the SU^2 solver run extended by the coupling-related, minimally invasive code changes to SU2_CFD in pseudo-code, which are necessary to integrate the coupling adapter *Precice* into SU^2.