UFR de Mathématiques et d'Informatique de Strasbourg

Training report M1 CSMI - BDR Thermea Group

DLL Importation and Exploitation

Quentin LOEB
M1 CSMI 2020-2021

Supervisor : DidierLEMETAYER

August 24th 2021

Remerciements

Je tiens à remercier dans un premier temps mon maître de stage Didier LEMETAYER, pour m'avoir accompagné depuis les débuts du projets jusqu'à ce jour, avec la constante volonté de me faire découvrir de nouvelles choses et de me faire progresser. J'ai pu réaliser un stage enrichissant malgré la crise sanitaire, en me sentant intégré à une équipe. De ce fait, mes remerciements vont également à l'équipe Simulation de BDR Thermea, ainsi qu'à l'ensemble des personnes que j'ai rencontré au cours de mon stage. J'ai pu passé de très bons moments en leur compagnie et me sentir intégré à leur équipe.

Je remercie également l'ensemble de notre équipe pédagogique qui, malgré la crise sanitaire, a su faire en sorte qu'on reçoive des enseignements dans les meilleurs conditions possibles. Cette année a été très éprouvante pour tous, mais ils ont su s'investir afin de nous éviter de baisser les bras.

Enfin, je remercie mes camarades de CSMI et tout particulièrement Marie REITZER, qui m'a accompagné et soutenu tout au long de l'année, et qui m'a permi de m'accrocher aux études dans les périodes compliquées.

Contents

1 Introduction

1.1 Presentation of the company

BDR Thermea Group is a leading european company (third in Europe) dedicated to the manufacture and sale of domestic and industrial heating appliances. Based in Apeldoorn (Netherlands), the firm was created in 2009 by the merger of several leading european brands (De Dietrich, Remeha ..).
BDR groups serve customers in more than 100 countries, employ 6200 people across the globe and has 1.8 billion euros in revenues.
BDR groups inovate to fight against climate change, it manufacture products with a near-zero carbon footprint and it developp innovative products to save energy and cut carbon emissions.

1.2 Context

Heat pumps are made of 5 major components produced by different suppliers : evaporator, compressor, condenser, expansion valve and a fan, linked to the evaporator. Each component is delivered with a DLL, that describes the characteristics and the functioning of the component. These DLL can be used in programms, and they can be included on Dymola, wich is a modelisation and simulation tool. We will work on the free version of Dymola, we could do the same work on OpenModelica, an opensource modelisation and simulation tool.

We will work on importing theses DLLs on Dymola, and make sure they communicate together in order to modelize a heat pump with all datas provided by the suppliers.

1.3 Roadmap

- Import a DLL on Dymola and verify the importation :
  - Make a simple C programm to import the fan's DLL
  - Use a simple fonction of this DLL in the programm
  - Study the different fonctions with the online documentation to understand how they work
  - Import the DLL on Dymola to modelize the component
- Import 2 DLLs on Dymola and allow them to communicate
  - Understand the physical interaction between the two components
  - Identify the useful functions and variables of the two DLLs
  - Import the second DLL on Dymola
  - Make the two DLLs communicate
- Modelize a heat pump on Dymola
  - Identify and study all the components of a heat pump and their interactions
  - Import all the DLL on Dymola
  - Make all the DLLs communicate

2 Generalities about DLLs

2.1 Definition

A Dynamic Link Library (DLL) is a library of functions that can be used in a programm. When included in a programm, a DLL is only called just before the execution of the programm (and not at the compilation). Most of the DLLs have the file extension .dll, and are almost only available for Windows. Thus, we will work on a Windows Environnment.

The major characteristic of the DLL is that it is a binary file, and we almost can't know anything about what is inside whithout a documentation. The DLL might be coded in many various languages such as Delphi, .NET, C#, C . . . The DLL importation will allow us to create a powerful tool that "reads" functions written in different languages, and use them together. It basically works as a black box, we put some datas as input, and we get output datas without knowing exactly the different steps of the functions. It isn't an issue in our case, since we only need to work with output datas.

2.2 The dumpbin.exe tool

How we said before, we almost can't get any information about a DLL without its documentation. Actually, the only tool we've found to get information from an unknown DLL is the dumpbin.exe tool[6]. It is a tool from Visual Studio 2019 that allows us to know the exact name of the functions inside a DLL. To use this tool, we have to open a command promp, and place ourselves in the folder "C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.25.28610\bin\Hostx64\x64". Then, we have to call the dumpbin.exe tool using the following command : "..exe /exports "path to the DLL". Executing this command will display a table with the names of the different functions inside the DLL. This tool could be useful if there is low or no documentation for the DLL, but doesn't bring that much information.

3 Dymola

3.1 Presentation

Dymola is a powerful simulation and modelisation tool developed by Dassault Systèmes, base on the Modelica language. It is used in several different domains such as automotive, aerospace, industrial equipment, fluid mechanics . . . Some of the work in this report can be achieved with the trial version, but we need a license to use advances components and libraries in order to simulate better systems. The equivalent-free version of Dymola is OpenModelica, also based on the Modelica language. This software is open-source and doesn't require any license. During the whole training, we worked on Dymola using a license.

3.2 Function importation on Dymola

Dymola has a lot of libraries with ready-to-use functions inside. It also allows us to create and use our own functions inside. As the Modelica language is quite basic, it is possible to import functions from an external source code (for example in C) and to call it directly in Dymola.
A good first exercise is to try to import a C function that computes the sum or two real numbers. As example, see how a simple function's importation works, step by step. We will create a file called addition.c and create our function, an example of source code could be :

```
    double addition(double x, double y){
        double sum = x+y;
        return sum;
    }
```

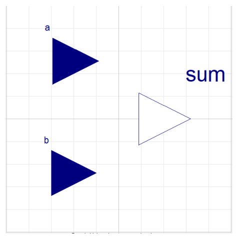

Figure 1: a,b are "RealInput", sum is "RealOutput"

The first step is to create a new function in Dymola. We have to set :

- The input variables and their types (Integer, Real...)
- The output and their types
- The language of programming
- The path to source code (being careful when using relative path)

To compile a code, Dymola works in a directory that is loaded at the launch. It is important to notice that Dymola will load by default the directory where it is installed. We can use the function "Modelica.Utilities.System.getWorkDirectory();" to know the path to the current directory. To set a path to another directory, we can use the function "Modelica.Utilities.System.setWorkDirectory("path");". Furthermore, we need to set a variable option in Dymola in order to tell if we want to compile in 64 bits or not. For the following of the work, we used the setting "CompileWith64 = 1", meaning that we won't compile in 32 bits. It is mandatory, as compiling in 64 bits will result in some errors.

The Modelica source code will be :

```
    function addition
        input Real a,b;
        output Real sum;
        external "C" sum = addition(a,b);
        annotation(Include="#include <addition.c>");
    end addition;
```

We used here a relative path, so the file addition.c should be stored in the current work directory.

The function is now ready, we have to create a component that uses this function. For each input parameter, we have to create an input block (here we will use Integer Input block), and the same thing applies to the output. Dymola works by connecting components together, and theses blocks will be the one we will use to connect together later. This diagram will create a source code for the component, and we just have to add two lines to tell Dymola how the inputs and outputs are linked each other. In this case, the output "sum" is the result of the function calculation "addition(a,b)", the syntax will be :

```
    equation
        sum = addition(a,b);
```

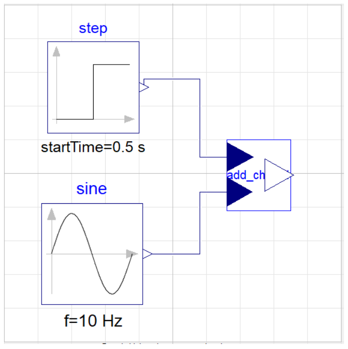

Figure 2: We define the Input parameters

The component is ready, we can now set values for a and b. They might not necessary be constant, there a plenty of mathematics functions in Dymola such as linear function, cosinus, step function . . . Here we chose a step functions, which equals -2 before 0,5sec and 2 after 0,5sec for the first input (a), and a sin function for the second input (b). Our system is now ready for simulation, we can simulate the model and plot the results in the "Simulation" window :

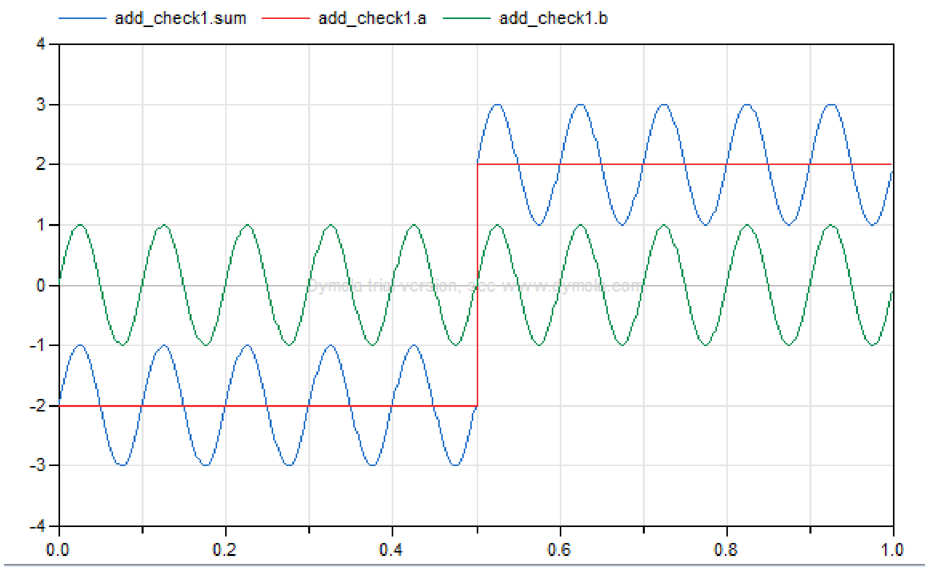

Figure 3: Results of the Dymola simulation

3.3 How to return multiple values in a function

With the previous example, we saw how to import a function that returns only one output value. In order to simulate more complex functions and more complex models, we would like to create C functions that returns several values. Unfortunately, using tables as function's output in C isn't practical at all to import the results in Dymola. Thanks to the technical note[2] written by Jean NOËL from the CETIAT , we get another more simple way to return multiple values from a C function in Dymola. It appears that if we give some pointers as input to our function, Dymola is able to read it's value after executing the function, as an output value. It is the easiest way we could found to have several output values in a function, as they are all independent in Dymola in contrary to a table.

Let’s work again with our example, and add a pointer that return the substraction. First of all, let’s add this pointer to our source code :

```
    double addition(double x, double y, double* substraction){
        double sum = x+y;
        *substraction = x-y;
        return sum;
    }
```

Then, let’s add the new variable "substraction" as an output in Dymola :

```
    function addition
        input Real a,b;
        output Real sum,substraction;
        external "C" sum = addition(a,b,substraction);
        annotation(Include="#include <addition.c>");
    end addition;
```

In the function definition, the new pointer is considered as an output, but is part of the functions variables. this matches with our function’s definition in the source code. Now, we have to update our component, and place a new output block for the variable "substraction". Once again, we have to update the related equation, following this syntax :

```
    equation
        (sum,substraction) = addition(a,b);
```

According to our component, we now have 2 inputs (a and b), and 2 outputs (sum and substraction), which matches our function definition in Dymola. The model is now ready to be simulated, since we don’t need to define the value of substraction in the simulation. We can repeat this process as many times as we need to add a new output value for our function.

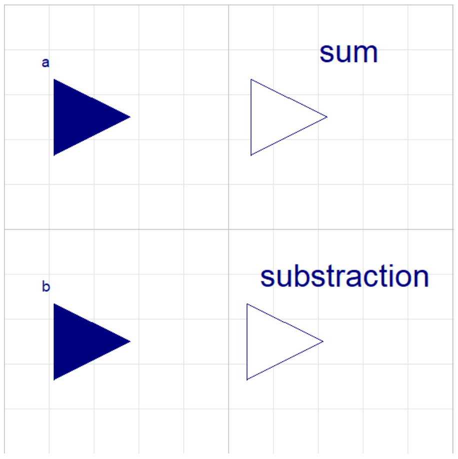

Figure 4: a and b are "RealInput", sum and substraction are "RealOutput"

4 Fan's DLL from EbmPapst

4.1 Presentation

The fan is a component that creates a forced air flow rate in order to improve thermal exchanges on the evaporator. We will work with the DLL of the fans from EbmPapst. Our first step is to import its DLL in a C program. They deliver a software called FanScout with the DLL, which allows us to validate the importation of the DLL and the functions by comparing the results of our program and their software. As said before, we are forced to work on Windows, and we have to include the <windows.h> package. We have to use the LoadLibraryA(char* "path") function to load the DLL at the execution of the program (not at the compilation). It returns a pointer of the library, which allow us to test if the program could load the DLL : if the pointer is a NULL-pointer, the operation failed. We also call the GetLastError() function, which gives us an error code if the operation failed, helping when debugging the program. We will print every message in a log file to improve visibility on the operations. It will mainly help us when running the function on Dymola.

```c
HMODULE ebmpapstFanDll = LoadLibraryA("../../data/EbmPapstFan32bits.dll");
if (!ebmpapstFanDll) {
    fprintf(log,"Error: Could not load the DLL (%d)\n", GetLastError());
    return EXIT_FAILURE;
}
```

In our case, two DLL files were available : one in 32 bits and one in 64 bits. We will only work with the 32 bits file, as the function in the windows.h library and also Dymola works better with this type of file.

4.2 First function importation

Our DLL is now successfully loaded, but we still can't use the functions inside. The different functions available in the DLL are explained in an online documentation, made by the supplier. We have to define new function, and use the GetProcAdress function, which retrieves the address of an exported function or variable from the specified DLL. We also have to define a new type that matches the skeleton of the function : in our case, a function that takes no parameter as input and returns and integer (the version of the DLL). The first function we will import is the GET_DLL_VERSION function, which takes no input, and simply returns the DLL's Version. Thus, it is a simple function to validate the importation of the DLL : if this function returns the same Version as in the FanScout program then the DLL is successfully loaded and we retrieve the good function.

```c
typedef int(__stdcall *GET_DLL_VERSION)();

GET_DLL_VERSION getDLLVersion = (GET_DLL_VERSION)GetProcAddress(ebmpapstFanDll,
                                                                "GetDLLVersion");
```

We can now try to run this function in a programm, and compare the results with the output in the FanScout programm. If the DLL's Version is the same, we know that the importation went well and that our programm and FanScout will return the same results if used in the exact same way.

4.3 Building the Fan calculation function

We would like to build a function which takes :

- The serial number of the fan

- The fanspeed [Hz]
- The air temperature [K]
- The air density []
- The pressure difference [Pa]

as arguments and returns :

- The air flow rate [m3/s]
- The electric power [W]
- The maximum fan speed [Hz]
- The average sound level [dB]

Input parameters First, we will need to define how we choose the different input parameters. Note that all the units we used are from the international system of units, so we will have to be careful since the DLL requires different units (for example °C instead of K).

The serial number will depend on the fan we will study. The DLL is delivered with a database containing datas for several fan models. Our function will have to look for the target fan in the database and load the corresponding datas. Each fan is associated to a unique key in the database, so we will have to search and find the matching key for the given serial number. This operation will need us to call the "GET_PRODUCTS_PC_", which returns the number of models in the database, and a buffer filled with each serial number and matching key. Then, we have to define the target fan speed we want to reach. The DLL functions have a "speedreserve" optional parameter, which is a percentage of the maximal fan speed, and returns the performances at the operating point at the target speed. To reach the target speed in our function, we first need to run the function without the speedreserve parameter to get the maximal fan speed with the conditions. Then, we simply have to calculate the speedreserve coefficient :

$$\text{speedreserve} = \left(1 - \frac{\text{target speed}}{\text{maximal speed}}\right) \times 100 \tag{1}$$

When running the functions with the speedreserve coefficient, it will now look for the operating point at the given target speed. Then, we will take basic values for the air temperature and density since these parameters will be only used later on, with the other components of the heatpump, we will set them to 298,15K (25°C) and 1,2.

The last parameter is the pressure difference, which is related to the friction between the air and the fan.

The function we will work with is "GET_CALCULATION_FAN_ALONE". It takes a string containing all the values as input, and a pointer to a buffer string which will be filled with the output values.

```
    typedef int(__stdcall *GET_CALCULATION_FAN_ALONE_PC)(char* in, char** out);
    GET_CALCULATION_FAN_ALONE_PC getCalcFanAlonePC = (GET_CALCULATION_FAN_ALONE_PC)
    GetProcAddress(ebmpapstFanDll, "GET_CALCULATION_FAN_ALONE_PC");
```

Warning : To create the input string, we might need to do some transformation operations. If we only print floats in a string, it will print the value with a dot as decimal separator. The DLL function only recognize commas as decimal separator, so we need to build a function to give the good format to our strings :

```
    void string_formatting_dot_to_vig(char* str){
        char v = ',', d = '.';
        for (int i=0; i<strlen(str); i++){
            if (str[i] == d)
                str[i] = v;
        }
    }
```

This way, our input string has the good format for the DLL function.

Output values Almost every function from the DLL takes a string buffer as input, and fills the buffer with all the output values. We can separate each value in this buffer using the strtok() function. This function allows us to get a copy of the string contained between to separators that we define (in this case, semi-colons). Before converting the string to double, we have to build a function just like before, to change the commas in dots, otherwise the conversion will return rounded numbers and we might lose some precision.

We now have all the tools to build a procedure for the fan, which will execute the following steps :

- Load the DLL
- Call "GET_PRODUCTS_PC_" and look for the matching serial number
- Call GET_CALCULATION_FAN_ALONE with required pressure, air temperature and density as parameter
- Return the maximum fan speed at the given operating point
- Calculate the speedreserve term to get the operating point at target speed
- Call GET_CALCULATION_FAN_ALONE with required pressure, speedreserve, air temperature, density and an empty buffer as parameters
- Return the electric power, air flow rate, and sound level.

The air flow rate will be the return value of our function, and the other values will be saved in pointers. The DLL returns the spectrum of the sound level for various frequencies : we will only calculate the mean of all these values since we don't need that much level of precision for this variable.

4.4 Fan's simulation

We can now create a new function to import the procedure for the fan, using the following code :

```
    function compute_fan
        input Real speed, air_temperature, air_density, required_pressure;
        output Real flowrate, electric_power, sound_level, maximum_speed ;
        external "C" flowrate = compute_fan(speed, air_temperature, air_density,
    required_pressure,electric_power, sound_level, maximum_speed);
        annotation(Include="#include <compute_fan.c>");
    end compute_fan;
```

Note that the serial number isn't part of the input values in Dymola : we didn't find out how to properly put a string as variable in the program. At the moment, we will only precise the serial number in the source code. Then, we create a component with 4 RealInput and 4 RealOutput. We define the input values and we're ready to simulate the component.

We will study the influence of the important parameters on the output. First, we are going to observe the air flow rate and electric power evolution at constant fan speed, and then do the same at constant required pressure. In all the simulations, the maximum fan speed was constant (since it's only based on the fan's characteristics), and the sound level was also almost constant.

On these figures we observe the influence of required pressure on air flow rate and electric power, at 4 different fan speed : 2000, 1800, 1600 turns per minute and at maximum performances. The flow rate is displayed in [m^3/h] for visibility purpose. Our first observation is that the curves don't cross each other and are parallel, meaning that every functioning point belongs to a single curve (so can be reached with only one fan speed). Also, the air flow rate decreases with the required pressure. We expected this results, as a higher pressure is harder to be "beaten" by the fan. Furthermore, the required pressure doesn't have that much influence on the electric power of the fan : as the pressure increased from 20 to 140 Pa the electric power only increased by 10%.

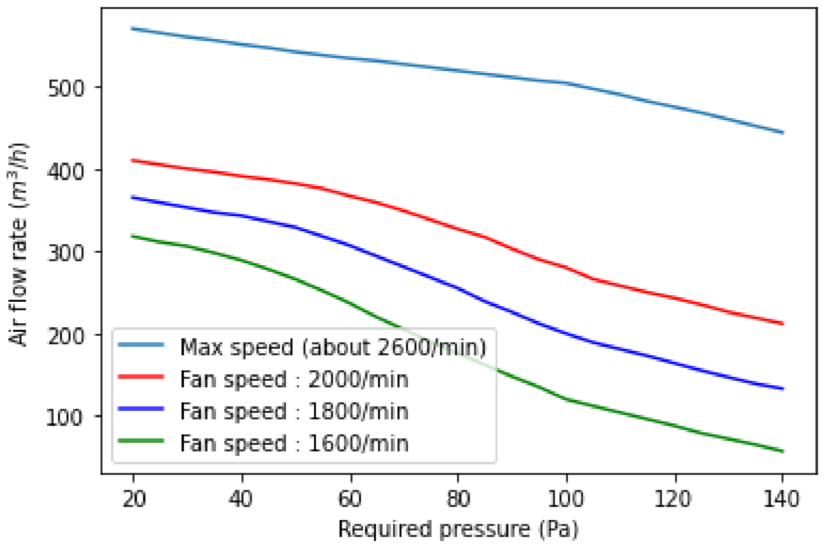

Figure 5: Influence of required pressure on air flow rate at constant fan speed

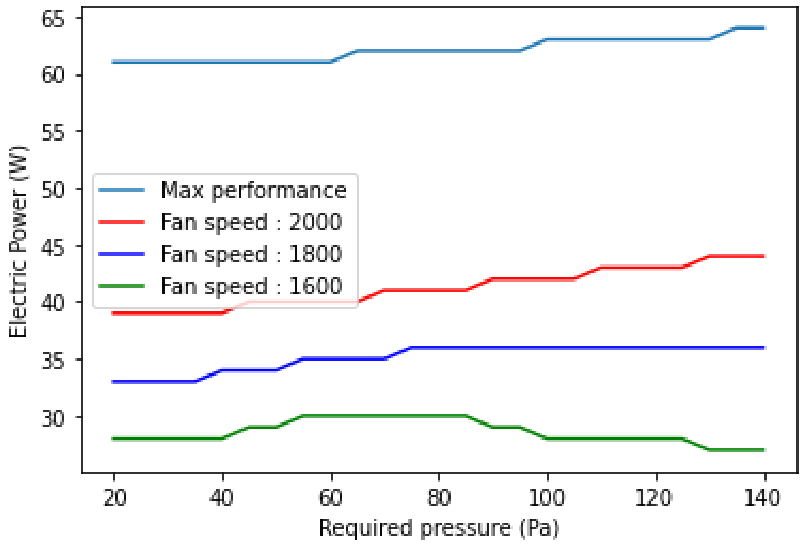

Figure 6: Influence of required pressure on electric power at constant fan speed

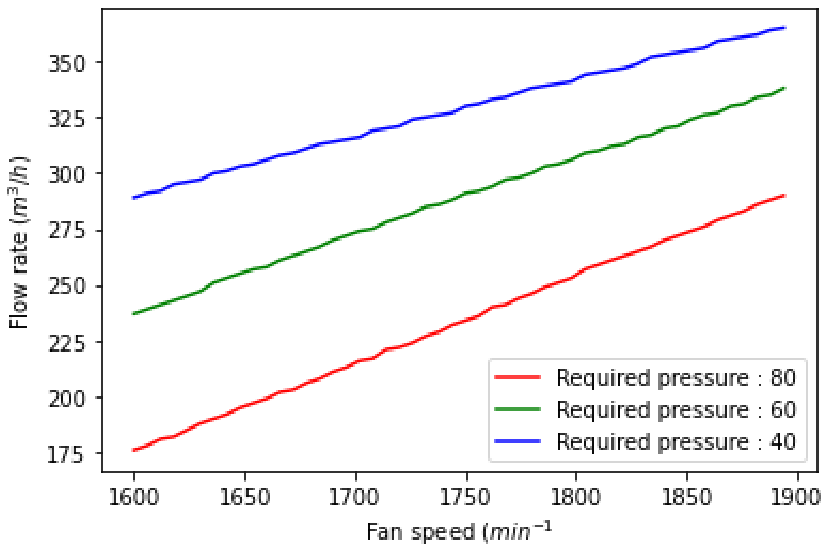

Figure 7: Influence of fan speed on electric power at constant required pressure

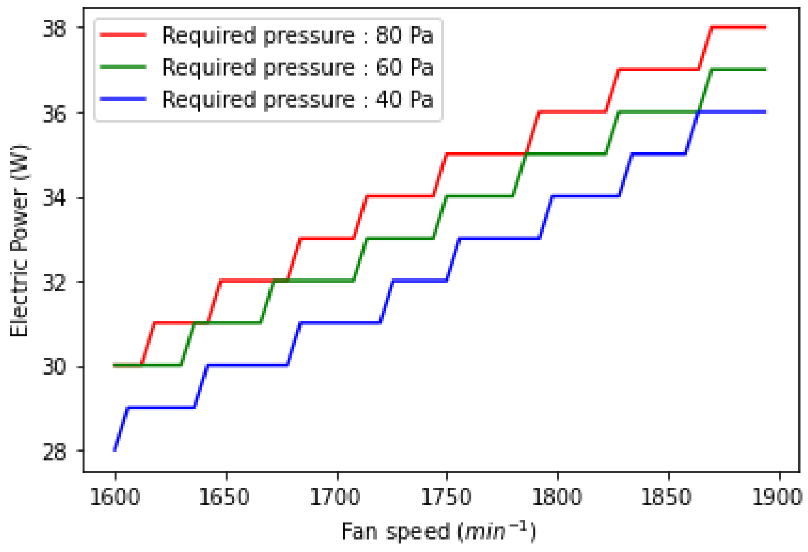

Figure 8: Influence of fan speed on electric power at constant required pressure

On these figures we observe the influence of fan speed on air flow rate and electric power, at 3 different required pressure : 80, 60, and 40 Pa. First of all, we confirm our previous observation : the air flow rate really decreases when required pressure increases. We also notice the linear relation between the fan speed and the air flow rate : when one increases, the other one also increases which is also an expected result. Moreover, the fan speed has a more significative influence on the electric power, they are also linked by a linear expression.

4.5 Integration of the built model in a Dymola model

Dymola already allow to simulate components juste like fan's, with generic parameters. The function we build can be integrated in one of this model, and allow to calibrate the model so that it fits perfectly to the chosen component.

The first model here is an evaporator and a fan built with components from Dymola.

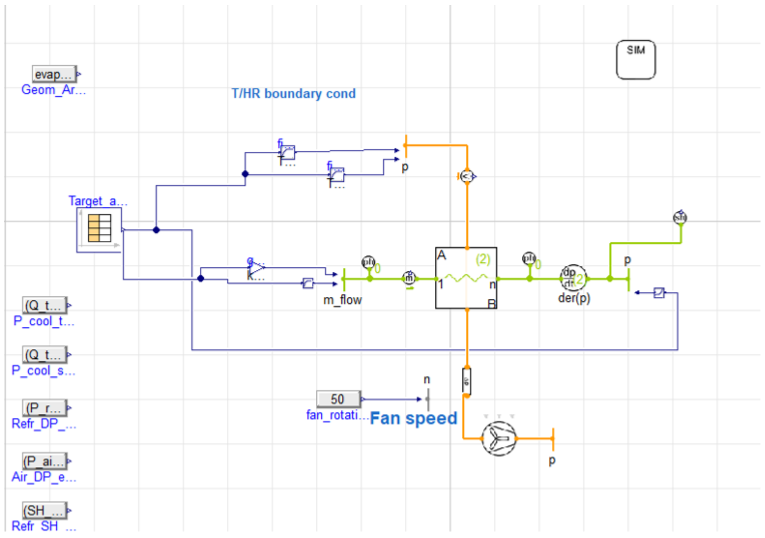

Figure 9: Model of evaporator and fan built with Dymola

We could add to this model the function that we built before to the fan, and impose and air flow rate as a boundary condition for the fan. The result will be a system calibrated to the exact model of fan we chose, since the function we added works with the DLL. This way, the DLL doesn't only have an influence on the fan, but also on the evaporator because the components are connected together. For the next steps, we will have to do the exact same thing with the other DLLs : build a base function, calibrate the model and link the models together.

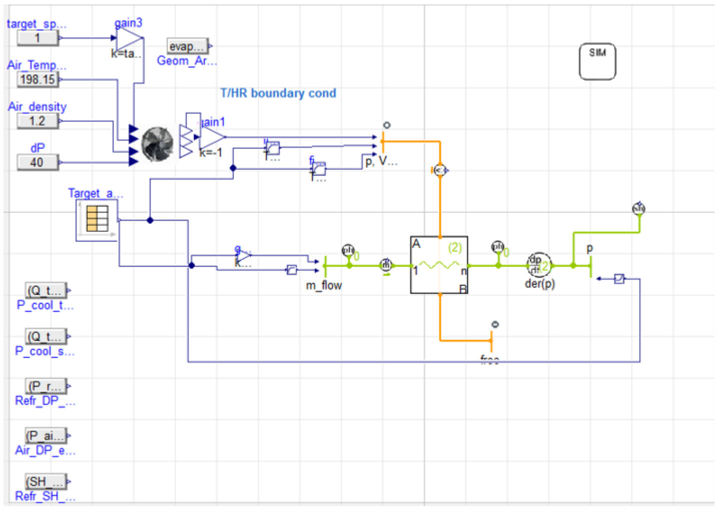

Figure 10: Model of evaporator and fan built with Dymola

5 Condenser's DLL from SWEP

5.1 Presentation

The condenser in the heatpump is the piece that transforms high pressur vapor into high pressure liquid in the circuit. The condensers from SWEP are also delivered with a DLL and a software juste like for the Fan. The DLL that is delivered is particular, since it doesn't look like the fan's DLL at all. When executing the dumpbin.exe command, we don't see any particular function inside. Fortunately, the DLL was delivered with a Visual Studio project to show us how it works. Actually, the DLL was written in .NET, and the initial importation was done in C. Then, a translation from C to C was made. Thus, this programm of high level complexity is really hard to understand for beginners and intermediate programmers.

5.2 Functions exploitation

After a lot of tries to import the DLL in a similar way as for the fan, we understood that the DLL wasn't built the same way, and couldn't be imported the same way. The only working solution after hours of researches and testing was still the delivered project. Unfortunately, there doesn't seem to be a way to create a function in Dymola importing a whole project. The only conceivable solution would be to create a function in a source code that calls the project and communicates with it in order to recover important informations. So the objective of this part will be to build a wrapper for the condenser.

First of all, we have to define the variables we want to control and the variables we want to observe. As inputs, we will have :

- The coolant liquid temperature at the start[K]
- The water temperature at the start [K]
- The coolant liquid pressure at the start[K]
- The coolant mass flow [kg/s]
- The water mass flow [kg/s]
- The number of plates in the condenser
- The type of coolant liquid

And as outputs, we will have :

- The capacity [W]
- The pressure difference for the coolant liquid [Pa]
- The pressure difference for the water [Pa]
- The water temperature at the end [K]
- The condensation temperature [K]
- The subcooling [K]

As the functions are already imported in the project, we don't need to do anything special to call them, they just work like regular functions.

Our wrapper function will take all the input as parameters and write them in a text file. This text file will then be read by the project in order to create the input string, and after calculation, the project will write the outputs in a text file. This way, our wrapper will save all the datas and it should be enough to import this on Dymola.

The remaining difficulty will be to execute the project without ending the wrapper function too soon. We tried to use the system() function to execute the .exe file from the project, but this always ended the current program at the call of system(). A solution would be to execute this command in parrallel, to make sur that the father's process will be able to get the values. As we are still working on Windows, we sadly can't use functions like fork() to create a new process. At the moment we're locked at this step, since we can't find a decent function to create a new process. Some functions like this seem to exist but there wasn't a way to used them that worked like intended.

6 Conclusion

Working with DLL isn't really trivial since it isn't a well documented topic. As we told in the beginning, DLLs can be implemented in many various languages, and it can be quite difficult to "convert" the work from a language to another. Without a clear and well detailed documentation, it is almost impossible to work on this topic. It is almost mandatory to have some examples programms that shows you how to do for each DLL, since we saw that there isn't an universal way to import them.

Furthermore, it can be quite hard to work with the C language because it is harder and less intuitive that some languages such as Python. There is really few room for improvisation for this project, and it can be really frustrating to be locked for many days and even weeks because of the few informations about the topic.

Additionally, Dymola is not really easy to work with at the beginning. There are really few indications on how things work, and there are even less indications when some problems occurs.

All of this makes the work on this project complicated, I spent and enormous amount of time trying to get the right setups on my computer and on the different software, and I spent even more time just trying to debugging all of the programs. It showed that it is mandatory to understand really well what's happening in the computer, and it also learned me to build my programms in such a way that I can know at every moment what when wrong. The work on this topic takes really more time than expected, since there are so many things to do before getting the first results.

However, it is a really interesting long term investment in time. When the model will be finished, the simulation will be really powerful, helpful and will take less time to do. Hopefully this will allow BDR Thermea to get better performances on their heatpump while saving a consequent amount of time.

References

[1] EbmPapst. *DLL (Online manual)*. 2014. URL: https://ebmpapst.atlassian.net/wiki/spaces/PS/pages/10256981/DLL+Online+manual.

[2] Jean NOËL. “Utilisation d’une DLL dans le logiciel Dymola”. In: (2020).

[3] sjoelund.se. *Is it possible to return multiple values from an external file to Dymola?* 2014. URL: https://stackoverflow.com/questions/20974365/is-it-possible-to-return-multiple-values-from-an-external-file-to-dymola.

[4] CodeGuru Staff. *DLL Tutorial For Beginners*. 2005. URL: https://www.codeguru.com/cplusplus/dll-tutorial-for-beginners/.

[5] Dr. Michael M. Tiller. *Modelica by Example*. URL: https://mbe.modelica.university/.

[6] Unknown. *Lister les fonctions d’une DLL inconnue grâce à dumpbin.exe*. 2020. URL: https://analyse-innovation-solution.fr/publication/fr/csharp/lister-les-fonctions-dll-inconnue.

[7] Unknown. *OpenModelica - Calling External C functions*. URL: https://www.openmodelica.org/doc/OpenModelicaUsersGuide/latest/interop_c_python.html.