Generation of Functional Mock-up Units from Causal Block Diagrams

Bavo Vander Henst

University of Antwerp
Model Driven Engineering
Bavo.VanderHenst@student.uantwerpen.be

Abstract

The purpose of this paper is to investigate the Functional Mock-up Interface. We will do this by exploring a method for translating Causal Block Diagrams, implemented in Python, to Functional Mockup Units. The process of doing this will be explained in the paper to better understand the difficulties and advantages of the FMI standard.
This paper will not only explain the generation of FMU's but will also look at the flattening and optimization of the CBD's before they are translated. By doing this we try to create a complete method for generating optimized FMU's from CBD's.

Keywords: Causal Block Diagrams, Optimization, Functional Mockup Unit, Co-simulation

Introduction

Models are increasingly used within software engineering to simulate the behavior of a system without having to explicitly code them. Models are widely used to prototype systems instead of explicitly code them, this is logical because they are less costly to make and more easily to adapt so we can try out different settings. One of the major problems with models is that they are made in a variety of different formalisms, each with their own purpose. This makes it impossible to exchange and use other models if they are made in an other formalism. The Functional Mockup Interface [1] tries to solve this problem by defining an interface for model exchange and co-simulation. In this project we will generate Functional Mockup Units based on co-simulation. To do so we will start from a well known formalism, Causal Block Diagrams, programmed in python. We will go trough the steps needed to go from the CBD implementation to the FMU.

We will start by investigating the Functional Mockup Unit in section 1. In section 2 the starting point of the project is explained. Section 3 will cover the flattening of CBD's, followed by the optimization step in section 4. In section 5 we will look at the generation of the FMU. The results of our experiments are described in section 6. section 7 concludes.

1. Functional Mockup Interface (FMI)

The FMI Standard consists of two main parts:

1. *FMI for Model Exchange*
2. *FMI for Co-Simulation*

The main idea of both interfaces is to create a standard interface for files so that different formalisms can be exchanged using this interface. In the following we will look at the two parts individually.

1.1. Model Exchange

The interface of model exchange consists of a set of functions that can be called by the solver to query the internal state of the model. By making the FMU we create a *black box* around the model which makes it possible to keep the internal structure secret. Because of the interface between the solver and the model we can replace the model with a newer one without having to change the solver. We can also exchange our model with someone else who uses an other formalism or solver.

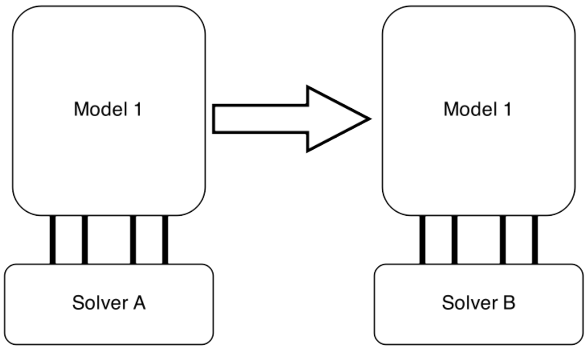


Figure 1: Visualization of the model exchange interface

1.2. Co-Simulation

The second interface, the co-simulation, is created to couple two or more simulators with each other. This can come in handy if we don't want to compile multiple models together in one, because we want, for example, to try out different configurations. An other possibility is that you don't have access to the source code of some models but only to the compiled versions. It is normal for a company to give out models of their systems in compiled form to protect their intellectual property. Here the interface isn't located between the model and the solver, but between the solver and a master program. This master program is responsible to synchronize all the simulations and to pass data between them.

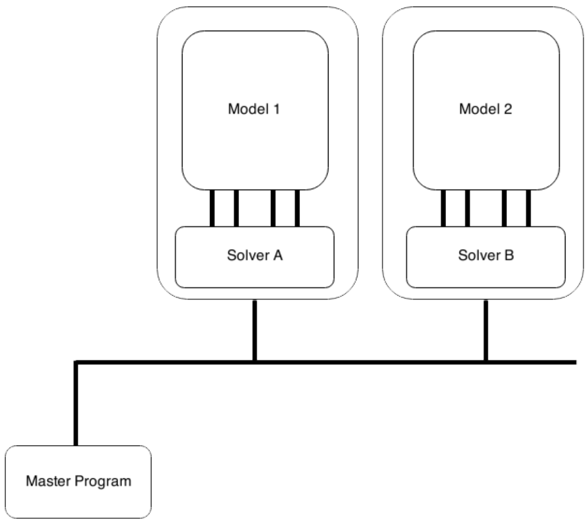


Figure 2: Visualization of the cosimulation interface

As we can see in the image, all the simulation kernels (model + solver) are connected to a bus. The master program as well is connected to the same bus. Now we can create a co-simulation environment where parts of the whole simulation can be swapped with new, better models.

2. Starting point

We start the project with a Causal Block Diagram implementation in Python. The python code gives a framework to create our own CBD. This framework is implemented in the file *CBD.py* and contains definitions for:

- **Base block**
  from which all the other blocks are derived
- **CBD block**
  which represents a CBD and can also be a hierarchical block in an other CBD
- **Linear blocks**
  (sum, product, negator, inverter,...)
- **Delay block**

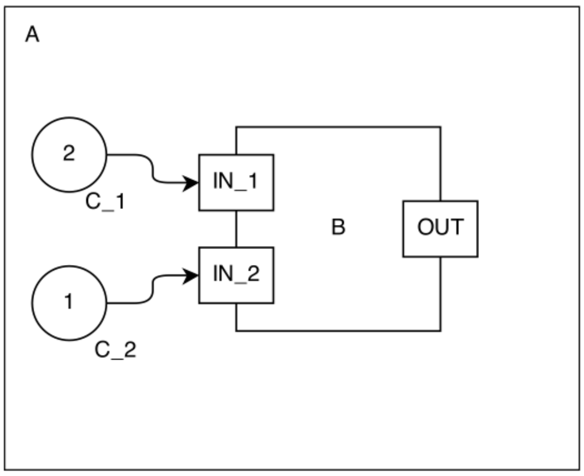
We add input and output blocks which are used as in- and outports for a hierarchical CBD block and adapt the function that connects the blocks. If the function is called with a hierarchical CBD block as argument, it will search in that CBD for an input/output block with a matching port name. The function will connect to that input/output block instead of the CBD itself. This way we can connect to input/output blocks form inside a CBD, but also from the outside which makes it possible for the implementation to fully express hierarchical CBD structures.

The second file that we already have is the *CBD-Simulator* class which implements a simulator for our python CBD model. The most important parts for us are the *dependency graph algorithms* and the derived *sorted list of components*. These parts, which are also used in the simulator, can be used to generate an order in which CBD blocks must be computed so that the input of the block is computed before the block itself.

3. Flattening

The first step in generating an FMU is to flatten the hierarchical CBD. By flattening the CBD we solve the difficulties that come with the hierarchical structure. This helps to make optimization and generation of the FMU easier.

We create the function *flatten* in *CBD.py* which implements the flattening algorithm. The steps of this algorithm will be made clear with a small example. Consider the flowing hierarchical CBD



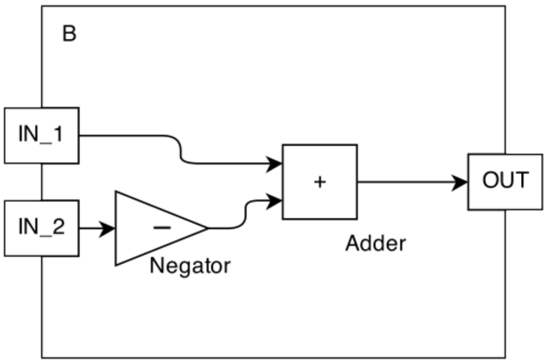


Figure 3: Hierarchical CBD

We have 2 CBD's: the first one (A) consists of two constants connected with another CBD (B). This one has 2 inports and one outport. The blocks inside the second CBD calculate the following formula: $IN\_1 - IN\_2$ by changing the sign of IN_1 and adding the result to IN_2.

The first step of our flattening algorithm is to copy all the internal blocks, of every CBD block, to the main CBD and than delete the hierarchical CBD block. All of the links between the blocks can be preserved because of the use of input/output blocks which allow us to connect blocks trough hierarchical levels. After this step one main CBD remains without hierarchy but still with input/output blocks.

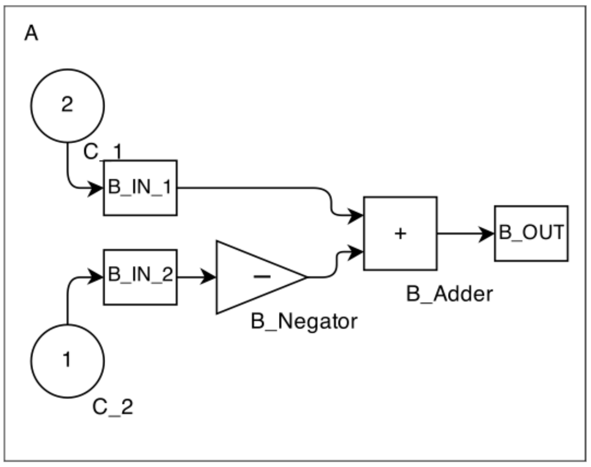


Figure 4: CBD with hierarchy removed

Note that the names of the internal blocks have slightly changed: they are prefixed with the name of the hierarchical block to make sure traceability remains possible.

The second step of our algorithm is to removing the redundant in- and out blocks. To do this we first remove all the connections between the input/output block and other blocks and make new connections between the input and their output blocks. This way all the connections stay the same but the in- and out blocks are removed from in between.

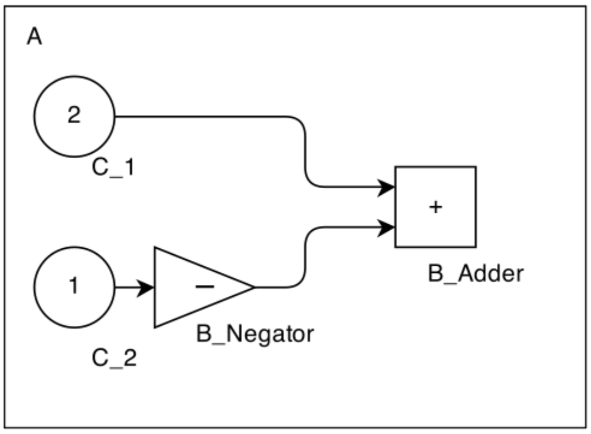


Figure 5: CBD without in and out blocks

Now we have a flattened CBD which can be used in the next steps.

4. Optimize

To make our FMU perform better we will optimize the CBD before the FMU is generated. There are multiple ways on how to optimize a CBD. Here, we will list a few of them.

4.1. Collapse blocks

The idea of this optimization step is to collapse a row of adder or product blocks together to one block. Because adders and products can have an arbitrarily number of inputs we can copy all the inputs of one adder, remove the adder and add those inputs to the inputs of the adder's successor. This isn't a real optimization, because it doesn't help removing any calculation, but it makes the CBD more memory efficient and it will make constant folding later on, more effective.

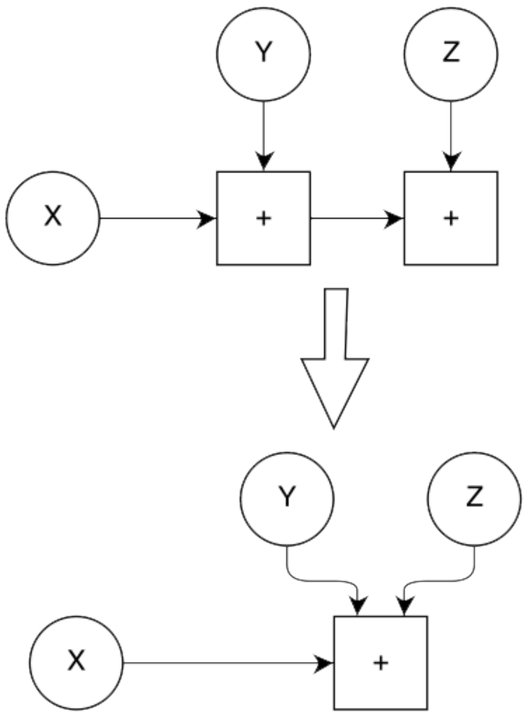


Figure 6: Example of collapse optimization

4.2. Constant folding

A second optimization is constant folding, where we will try to find lineair blocks with constants as input. These blocks can then be replaced by the computed constant, resulting in less redundant calculation during every time step.

This can also be a partial constant folding as it is possible that, for example, an adder has 2 constants and one non constant. The adder won't be replaced by a constant but the 2 constants can be summed, and the result will be added as a new constant block to replace the 2 constant blocks. This way the adder has less input that it must sum.

4.3. Special cases

There are still a few special cases of constant folding that can help speed up the CBD.

- A constant 0 as input for an adder can be discarded because it doesn't affect the outcome of the adder.
- A constant 0 as input for a product will make the whole product 0, so we can replace the product by a constant 0.
- A constant 1 as input for a product can be discarded because it doesn't affect the outcome either.

All these methods help reducing the CBD and removing redundant calculations.
All these optimizations use the dependency graph to optimize in the same order as the blocks are calculated. If we don't do this, some blocks will be optimized before their inputs are, which can affect the optimization process.

5. Genereate Functional Mockup Unit (FMU)

To create an FMU we need two files, as explained in [1], a C-file and an XML-file. We will discuss both files separately, together with their generation from our CBD. We start with the XML-File

5.1. XML-File

The purpose of the XML-file is to give information about the FMU. The file contains 2 major parts. The first one is the model description which contains the information about the FMU like the version used, the name of the FMU, GUID, ... The second part is a list of model variables, which are the ones you make public for the user of this FMU. These model variables contain some extra information like their reference, type, ... If we are using co-simulation, a third part is added, the implementation details about the solver that is used.

The generation of the XML-file is fairly straight forward. The model description is filled in with the information of the CBD. A random GUID is also generated and filled in. The next section in the XML-file contains the model variables, for those we loop over all our blocks and for every block we create a scalar variable. These variables are filled with the right name, value reference and type.

We use a SDK [2] to help us build our FMU's to ease the work needed to be done. One of the main advantages of the SDK is that we don't have to create separate files for co-simulation and model exchange. For co-simulation the SDK will automatically append the extra implementation details needed.

5.2. C-File

The C-file must contain a set of functions that can be used to query the state of the model/simulator and to communicate with the simulator.
The SDK is a big help here: with the SDK comes C-code which implements most of these functions for us. The only thing we have to do is to create a set of functions in our own C-file which will include the C-file from the SDK.
In our own C-file we must maintain the internal information of the converted CBD together with the calculations that are needed to be done every time step. The file starts with some defines required by the SDK like the number of reals and our GUID.

The next part of the C-code is the definition of the reference value of all our CBD blocks. By doing this we make the code more readable and neat without affecting the speed of our program. Note that these reference values must be the same as in the XML-file.

To hold the state of our FMU we create an array with enough space to save the current signal output of every block. Although a matrix can hold the signal output for every time step we have chosen for an array for two reasons. The first reason is space, the matrix takes a lot more space which will lead to a slower calculation. The second reason is that, at compile time, there is no way in knowing how many time steps we will have to compute and we have no means to create a sufficiently large matrix. It is however no problem if we want to collect data from every time step, because the master algorithm can query the FMU after every time step and save that signal data.

The rest of our file contains our functions that will be used by the SDK later. The first function is the *setStartValues()*,which is called on startup of the FMU to init all the variables. To fill in this function we run our python CBD for one time step so we can use the calculated signals. In the *setStartValues()* function we fill our array with the signal output of the first time step for every corresponding block. It is needed to init the array with our CBD's first step because the FMU standard see the initialization as the first time step.

The second function is the *getReal* function, this receives a value reference and returns the right signal value from our array. This function is pretty straightforward and doesn't need any more explanation.

The last important function is the *computeVariables* which will be called by the *fmiDoStep* of the SDK. In this function we update all the variables to the value of the next time step. To do this we loop over all the blocks of our python CBD starting with the delay blocks. We need to start with these blocks because they need to contain the value of the previous time step. Because we only remember one time step we must first copy the info of the previous time step to our delay block before that value is overwritten with the next time step. After the delay blocks the formulas of all the other blocks are printed to the C-file. This happens in the order of the sorted list of components so that the input of a block is evaluated before the block itself.

There is one exception to this process, namely the strong components. The internal blocks can't evaluated one by one because they depend on themselves in a circular way. Therefore we must calculate the results of all the blocks together, we do this by solving a system of lineair equations. For every strong component we create two matrices to represent our lineair equations, fill them with the signal values from the current time step and call a lineair solver to solve the system. When the lineair equations have been solved, we have the new signal values and we can continue with our function.

Note that because we have an array and no matrix, their is no need to re-evaluate constant blocks which gives us a speed up.

6. Experiment

To evaluate the conversion we propose in this paper, we will investigate a *real-life* CBD, implementing the pitch control of an F14 fighter. This CBD is translated from a ®MathWorks ®Simulink implementation to our own CBD format, and then transformed to an FMU. The translation from MathWorks to CBD is outside the scope of this paper.

To ensure the correctness of the conversion we compare our results with those of Simulink. The calculated *residual sum of squares* equals $5,40823E^{-20}$, which is acceptably small.

To investigate the effectiveness of our optimization algorithms, we compare the mean computation times of two CBD's: one without optimization and one with optimization. We will do this for the python CBD and for the CBD converted to an FMU. The results can be found in the following table.

F14 Model		Mean computation time (s)
Python	Not optimized	9.96355
	Optimized	8.570786
FMU	Not optimized	0.012619
	Optimized	0.009318

The optimization gives us a speedup of almost 16% for python and even 35% for FMU because there are less blocks to be computed, more specifically: there are 140 blocks in the F14 model, but after optimization there are only 116 left of them.

If we look at the difference in computation time between the FMU in comparison with the python implementation, we can see that the FMU is substantially faster. This is mainly caused by the difference in program language. The programming language of an FMU is C, which is faster than python in doing mathematical calculations. Moreover, the python implementation was not created with speed in mind whereas the FMU framework and our own code were.

7. conclusion

In this paper we have tried to purpose a method of forming Functional Mockup Units from Causal Block Diagrams, not only focusing on the translation itself, but also on the preparation and optimization of the CBD. As we have seen in our experiment, the conversion to an FMU caused a substantial speedup compared to the python CBD. Also the optimization helped reducing the computation time. We can conclude that the conversion from CBD's to an FMU is possible and that the FMU can be a good choice to be used instead of the CBD in a co-simulation environment.

References

[1] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauss, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, et al., The functional mockup interface for tool independent exchange of simulation models, in: Modelica'2011 Conference, March, 2011, pp. 20–22.

[2] QTronic, https://www.qtronic.de/en/fmusdk.html, fmu sdk.