Visual modelling environment for CBD's

Michaël Deckers
20145715
University of Antwerp, 2014-2015

Abstract

Causal Block Diagrams are a powerful, visual and easy to use way of modelling complex mathematical expressions or systems. To study them in the MoSIS course, one could only use textual code, which provided no visual feedback and almost no straightforward debugging. I set out to design a modelling environment in AToMPM that provides all of these things. Part one of this paper contains the results of research that has been done on the subject of debugging CBD's and AToMPM. Part two gives a detailed report on the design of the visual modelling environment for CBD's

Keywords: Causal Block Diagram, CBD, AToMPM, model debugging, CBD debugging, AToMPM debugging, breakpoint

Part I
Reading assignment

1. Introduction

Causal Block Diagrams (CBD's) is a visual modelling language that allows the modelling of mathematical expressions, including boolean algebra (and, or, not). The main building element of a CBD is a block, that contains either a value or an operator (+ , - , * , root, power , AND , OR , NOT ...). A collection of blocks is connected through signals, which connect the output of one block to the input of another. Most blocks have a particular set

of inputs (and outputs). By simulating the model, all blocks (except value blocks) perform their operation on their input(s) and send the result to the output. This paper does not try to explain CBD's in detail, and during the rest of this text, I will assume that the reader is familiar with the concept of CBD's. For more information one could start by reading the background in (Vangheluwe et al., 2014).

Although the basis of this language is the fact that it is visual, no modern and easy to use tool exists to create CBD's and, more importantly, run and analyze them. An effort was made using Atom3 (Vangheluwe et al., 2014), but this never gained much success. With this project, I hope to build a more solid and more functional modelling environment, using the tool *AToMPM*. An important aspect of the assignment is to be able to debug an implemented CBD, an issue that will be the main focus throughout the reading (first) part of the project, and a major part of the implementation (second) part.

This paper will start with an introduction to the reading/research material that was provided to me, on which I will base my findings and assumptions in the rest of this paper. (Section 2), it will then continue with a very short piece about CBD modelling in AToMPM in section 3. After that, I will talk about CBD debugging in general in section 4. A short notice about time follows in section 5. Section 6 will briefly introduce the debugging system in AToMPM and section 7 talks about CBD debugging with AToMPM and python. Section 8 concludes.

2. Introduction of research material

The basis for this research were two papers: "Explicit Modelling of a CBD Experimentation Environment" by Vangheluwe et al. (2014) and "Explicit Modelling of a Parallel DEVS Experimentation Environment" by Van Mierlo et al. (2014). Both of these papers were (at least in part) written at and by professors and students from the University of Antwerp. The focus in both papers lies on a developing a (modelling) and, more importantly, debugging environment for CBD's and parallel DEVS respectively.

3. Modelling CBD's

The main part (although not the difficult or time-consuming part) of this project is to implement a modelling environment for CBD's in *AToMPM*. Doing this is rather trivial and does not need any particular research. The

instructions that I have received from the classes and assignments on both CBD's and AToMPM should provide me with all the tools and knowledge I need to create the models and formalisms that are required by the project. This allows me to focus my research on the more challenging part of the project: debugging CBD's.

4. CBD Debugging

Building a visual CBD modelling tool that includes a debugger can greatly improve the efficiency with which diverse CBD's can be developed, since it is ofter very hard to manually pinpoint an error during the execution of a CBD, especially when the execution is traceable only in a textual way.

Before we start modelling a CBD debugger, let us first examine how CBD's should be debugged. The CBD simulation main algorithm (see listing 1) contains two nested loops. One is the `while` loop at line two, the other the `for` loop at line four. These two loops are the main points of debugging, the two places where a breakpoint[1] might be placed and triggered.

Listing 1: CBD simulator main loop (Vangheluwe et al., 2014)

```
time step = 0
while not end condition do
    schedule = LOOPDETECT(DEPGRAPH(cbd))
    for gblock in schedule do
        COMPUTE(gblock)
    end for
time step = time step+1
end while
```

4.1. Breakpoints

4.1.1. (Big) step

So what does each of these steps do, and what can be achieved by placing a breakpoint in an iteration of either steps. First, let us start with the larger `while` loop. As you probably know, the state of the CBD (when it has

[1] A breakpoint is a condition in the execution of a program or simulation which, when it evaluates to true, halts the program. This condition can be as simple as getting at a particular statement in the code.

some notion or time, discrete or continuous) and its blocks and signals can be calculated over time using multiple iterations. The `while` loop of the algorithm defines these iterations: every time the `while` is evaluated to true, the global state (all blocks) of the CBD is one step ahead, every block has made a calculation and is updated. Putting a breakpoint at this level allows the modeller to evaluate the workings of the CBD on a scale where it appears that each iteration, all blocks are updated instantaneously and at the same time. This step is not the most detailed, but nonetheless very useful. In the remaining part of this text, I will refer to this type of step as a *Big step*, or simple a *step*.

4.1.2. Small step

In a Big step, it appears as if everything happens at once, this is of course never the case in algorithms, and CBD's are no different. That is where the nested `for` loop comes in. During the methods `LOOPDETECT` and `DEPGRAPH`, a particular order has been given to the blocks of the CBD. The calculations of the blocks are performed in this order. The calculation of a single block in a CBD is what I call a *small step*. Putting a breakpoint in between small steps allows the modeller to find errors in connections between or the order of blocks.

4.2. Execution steps

Of course breakpoints are not the only way of debugging an algorithm. Sometimes it is easier or faster to just execute each part of the algorithm manually and evaluate the state of the CBD every step. To implement all possible ways of execution Vangheluwe et al. (2014) have introduced four different methods of execution. I will list them below but adapt some of the names so they fit my terminology better:

- **auto** or simply **run**: The simulation is ran normally, with a given time between iterations. This time between iterations allows two things:

  1. The modeller can monitor the state of the CBD during execution.
  2. The modeller can pause the execution of the simulation at a particular time, when the CBD is in a particular state.

  During a normal *run*, the presence of breakpoints might also cause a pause call and halt the execution. When the execution is halted, the user can inspect the state of the CBD more thoroughly and then

continue running, going step by step (see next bullet point), or even running as fast as possible (see third bullet point).

- **Big step** The CBD is ran manually, each click results in a big step advance in the CBD. The simulation is halted again after one step.
- **small step** The CBD is also ran manually, but now each step results only in the next block transition or calculation.
- **as fast as possible** The entire CBD simulation is ran *instantly* (limited by computational power of the computer). The user does not have the option to halt and breakpoints are skipped over. After execution the only debugging result is a trace of the system throughout all its states.

5. Time

Although the notion of time is discussed in great detail in both papers, I do not believe that having different notions of time has a major effect on the debugging or simulation experience when it comes to CBD's. The only *time* our simulation will need is the speed with which it executes its steps in the *run* execution mode, which can easily be modified by the modeller should he desire.

6. Model debugging in AToMPM

A big challenge for debugging a running CBD simulation will be linking the (empty and *dumb*) AToMPM execution and graphics to the back-end, which exists of a python CBD simulation script that contains all the information and calculations needed to run a CBD simulation. Two major difficulties will need to be overcome to implement this: First, I will need to compile the AToMPM model into python code and send this to the python script, so it can be ran. Second, during execution I will need to report the results back to AToMPM, yet still allow AToMPM to control the execution (halting, step by step, breakpoints, ...).

Figure 1 shows a very simplified but accurate model of how this is going to work: Clicking a button on the toolbar[2] will send a request to the translator/exporter. This request represents one particular command. The request

[2] Which is yet to be developed. The toolbar will contain all the buttons for running, pausing, steps, ... the simulation.

is than forwarded to the python simulation. There, a reply is generated from the logic the simulator holds. This reply is sent back to the translator, where it is mapped to an edit request for the model in AToMPM. This form of implementation is based on the one suggested for parallel DEVS debugging in Van Mierlo et al. (2014).

I will personally be responsible for developing the top and bottom layer (AToMPM and python simulation). At this time I do now yet know exactly how I am supposed to implement this link, more research and conversation will be necessary before this linking part can be implemented. The middle part will be developed together with my supervisor, since this is beyond the scope of this project or the course this project is part of.

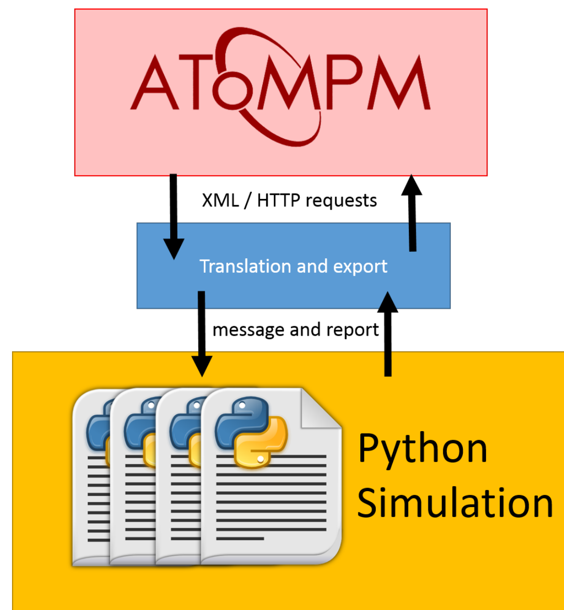

Figure 1: Simplified CBD debugging architecture

7. CBD debugging in AToMPM

In order to be able to use ways of debugging, we need to provide AToMPM with the right tools. Further in this section, breakpoints are discussed in detail. Step by step execution is not issue we discuss at this time, since I have no real knowledge at this time about how it is going to be implemented and because this has no real effect on the form of the CBD model.

7.1. Breakpoints

Like presented in Vangheluwe et al. (2014), a good way of adding breakpoints is to simply generate an extra CBD-type block, named an *halt* or *breakpoint* block, which can trigger a simulation halt when it it is computed by the simulator. For this to work, the block will have to be included in the dependency graph of the CBD. This means that the python logic has to be modified in order to work with this new type of block. The specific

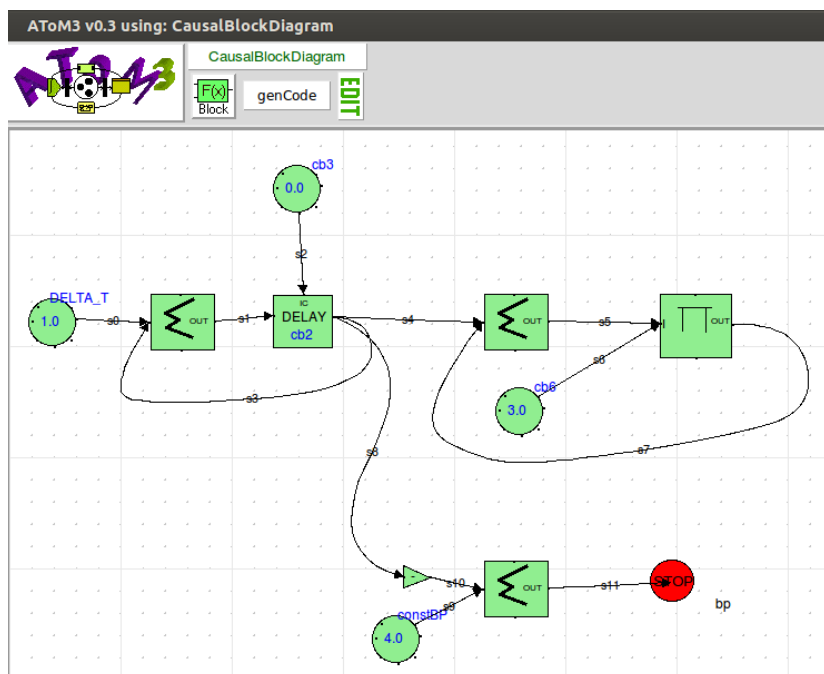

Figure 2: A CBD with a breakpoint block (red) modelled in the depricated AToM3 environment (Vangheluwe et al., 2014)

form of this block is still open for research. Either a block with only an input and no output, like shown in figure 2, or one with both an input and an output. In my eyes, the second option seems better, this way the block can be put inbetween two blocks, resulting in no need for additional blocks or non-determinism, which can exist if both a breakpoint block and another block are connected to the output ports of one block. Using an *in-between* block enforces a correct execution of the CBD, because it eliminates any difficulties in the construction of the depency graph. However, this is all just

my theory and I will have no proof of this until I implement and compare both solutions.

8. Conclusion

Using just two rather concise papers, I was able to get a decent idea about how model debugging (Causal Block Diagrams in particular) can be implemented for the graphical modelling tool AToMPM. Translating the notion of theoretical model debugging onto a graphical and working simulation is challenging, but not impossible. Using the CBD modelling methods from Vangheluwe et al. (2014) and the AToMPM debugging solution from Van Mierlo et al. (2014) as a base of research and combining this with my knowledge of CBD's in general, I was able to produce some information on the following topics:

- CBD debugging can happen two particular ways, namely halting the simulation and inspecting the values at runtime or examining a trace of all states of the CBD post-run.
- The first way of debugging can be done in two different ways: using breakpoints, which halt the execution of the program when a certain state is reached, or using manual steps, where the modeller is able to pause the simulation and continue the entire or part of the execution manually, using either *Big steps* (one time step or simulation iteration at a time) or *small steps* (one block at a time).
- Using breakpoints in CBD's is possible by adding a new *breakpoint* block that induces a halt in the simulation when it is solved.
- I will need to create link between the visual AToMPM environment, which is in fact just a "dumb" shell and has no knowledge of the internals of a CBD, and a CBD simulation environment, which contains the internals of CBD's, but no visual shell and is implemented in python.

Part II
Implementation

9. Introduction

In the reading assignment, I have tried to discuss and list the (theoretical) debugging methods for Causal Block Diagrams, but this is only a smart (be it difficult) part of this project. What was requested in the first place is a visual modelling environment for CBD's. Being able to simulate and debug this CBD is a part of this, but not everything.

The implementation part takes what I have learned during the reading assignment and tries to make this a reality. Of course I have to start with building a formalism so I can actually design a Causal Block Diagram. The creation of this CBD formalism is discussed in Section 10. The modelling environment was, in principle, done, but it would only make sense to use it if it also has some deeper functionality, so I set out to design a simulator as well. In order for this to work I have to export my model somehow. In agreement with my supervisor I decided to export the model to MetaDepth first and then compile it into Python, the language of the simulator. This is all discussed in more detail in Section 11. Once this is finished, I can build the simulation back-end and connect it to AToMPM. My supervisor provided me with a similar solution used for ParallelDevs simulation, which I could modify and extend this so it would work for CBD's, this way there was no need to do too much research or do unnecessary work. The final solution uses Statecharts as the main simulation tool. All the details can be found in Section 12. Any future work I can think of is discussed in Section 13. Section 14 gives an overview of all the files that have some meaning for this project. Section 15 concludes.

10. The AToMPM syntax

The first task I had to perform was developing a syntax in AToMPM which allows users to create visual CBD models. As with any syntax model, I started with the abstract syntax and after that created the concrete syntax. The process of, and the ideas behind the design will be explained in this section.

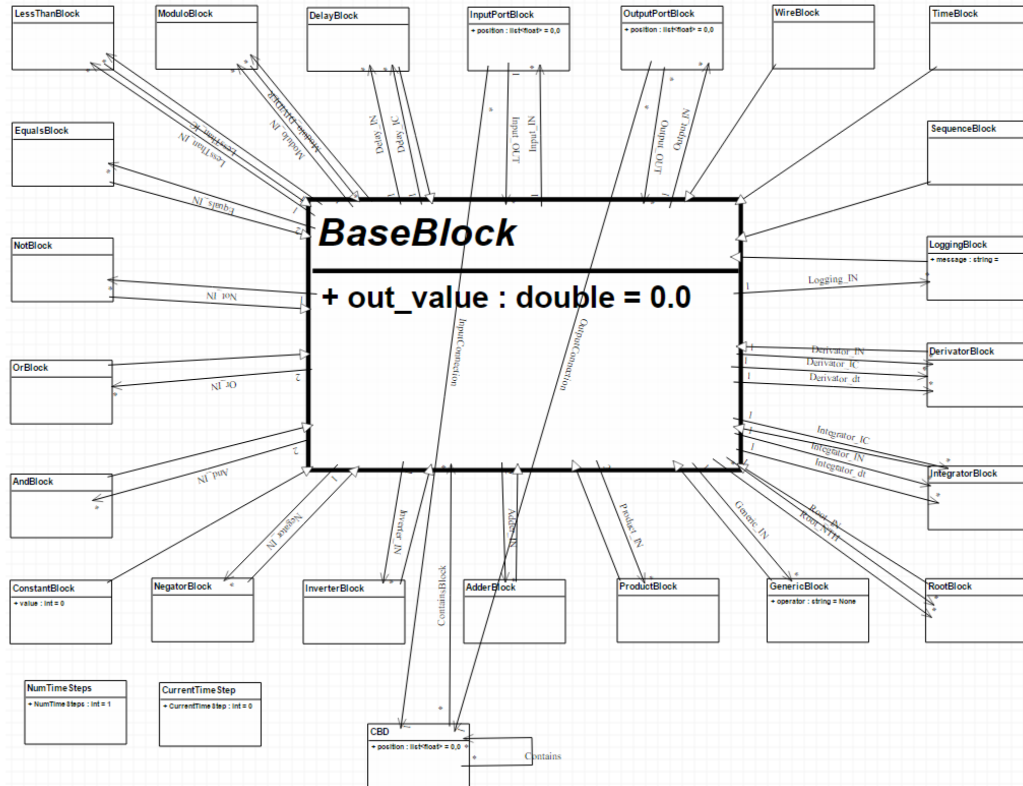

Figure 3: The abstract syntax model in AToMPM

Figure 3 shows the (UML) model of the abstract syntax. I have tried to base the implementation on the existing CBD simulation file that was used in previous assignments of another course (Modelling of Software Intensive Systems), which will be used for simulation (more on this will follow below). This meant that every type of block you can see in the image (the blocks in a rectangular shape around the BaseBlock class) all types of blocks inherit from the BaseBlock type. The only real difference between this model and the existing simulator is that the CBD class itself does not inherit from the BaseBlock instance.

Using inheritance it is possible to connect each block to another one without having to add lines to every single other class of blocks in the abstract syntax model. Another advantage is that the `out_value` attribute is part of every type of block. This attribute is only used for graphical reasons: this is where,

during simulation, the current value of the block is shown to the modeller. The value has no further impact on the simulation itself. This also means that while building a model, it is not necessary to change the `out_value` since it will just be overwritten.
The association links between the different types of blocks and the `base_block` show (and constrain) the incoming connections of other blocks. For each type of connection (e.g. `IN, IC, dt, ...`) the link also has an incoming cardinality, which limits the amount of incoming arrows of that type. For example: the `AdderBlock` had one link named `Adder_IN`, but this link had cardinality 2, which means that this block must have two inputs of the same type. If it is important which input sits on which side of the block operation (= commutativity), for example in the `LessThanBlock` or the `RootBlock` than two inputs of a different type must be used.
An instance of the CBD class can contain all instances of all types of blocks and even other instances of the CBD class, this way it is possible to nest multiple CBDs. (Currently it is only possible to implement nesting with a depth of one, meaning one extra level of CBDs within the main CBD, this is not a limitation of the model, but rather one of the simulator, the development of which was not part of this project assignment.)
The remaining two classes that can be found in the abstract syntax model are named `NumTimeSteps` and `CurrentTimeStep`. The first one the modeller can modify to tell the simulator how many steps should be simulated. The second one should (just like `out_value`) not be modified: this one shows the current step while the simulation is running.

10.2. Concrete syntax

Designing the concrete syntax model was not particularely challenging. I started with creating images representing the different types of blocks. Each class of block got its own color: yellow for boolean operators (smaller than, equals, NOT or AND, ...), green for mathematical operations (addition, multiplication, root, derivative,...), blue for constant or time blocks (no operation), orange for the delay block, purple for the Logging block (which does not really have much use in the current implementation and simulation), and grey for the Generic block. All of these types are shown in Figure 4. Two special blocks are the `InputPort` block and the `OutputPort` block, which serve as a connection between blocks in different (but nested) CBD's. Additional syntactic elements are the CBD itself and one for both the total and current time steps of the simulation. I did not implement the sequence

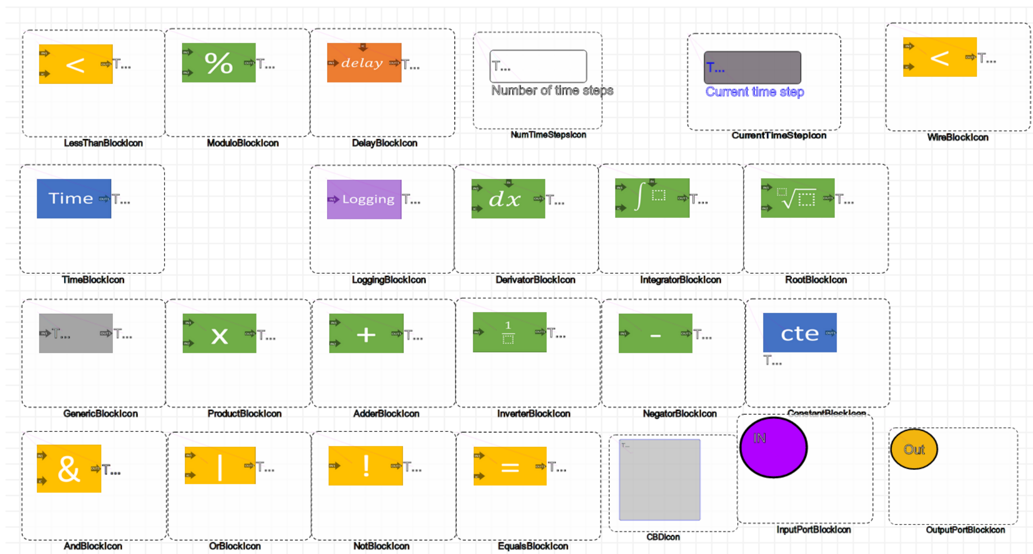

Figure 4: The graphical view of all the blocks and other classes in the abstract syntax model

block because it should only be used for testing purposes and using it should be avoided.

Of course in order for a CBD to work, blocks have to be connected so signals can be sent. Each type of connection gets its own color: black for an `IN` connection, blue for an `IC`(initial component) connection or a block-specific connection (nth root, divider), red for a `delta_t` connection. Containment links (between CBD and block or CBD and CBD) are transparent, as well as the links between a CBD and its input and output ports. Incoming connection to an input and output port have no arrow head to signify that these are just pass-through blocks. The results can be found in Figure 5

10.3. CBD formalism

When both the abstract and the concrete syntax are compiled the modeller can import the CBD toolbar as shown in Figure 6. Using it works exactly the same as every other formalism in AToMPM so no further discussion should be required here.

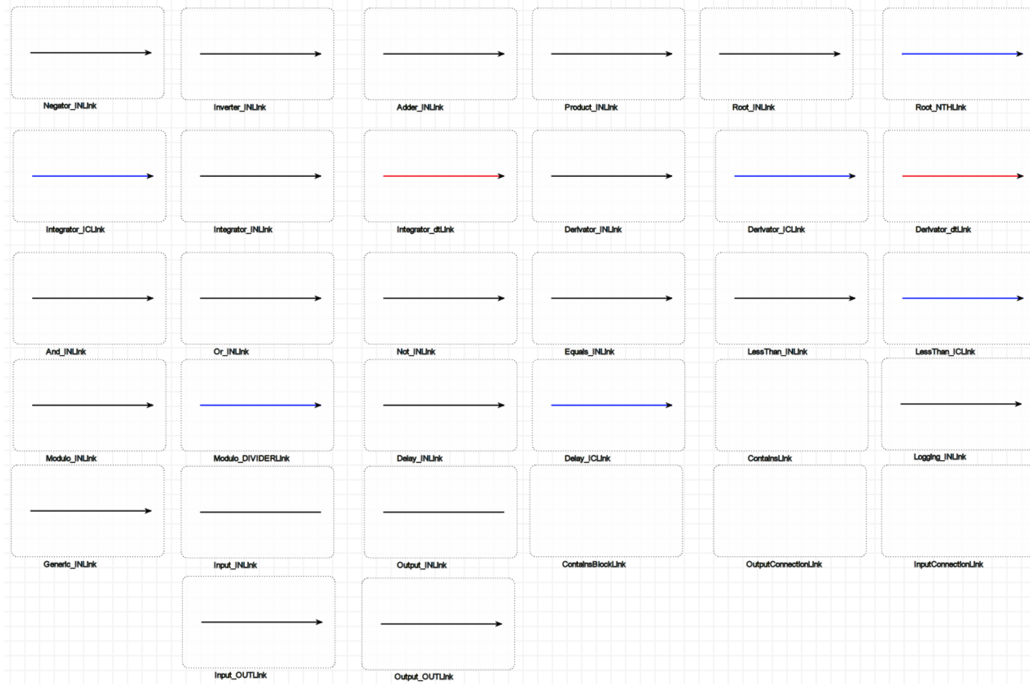

Figure 5: The graphical view of all sorts of links

Figure 6: The resulting toolbar used to actually design CBDs

11. Exporting to Python

The existing simulator that will be used to simulate and debug the model is written in Python, this means that the, in AToMPM designed, model will have to be converted to python somehow. To do this I have used MetaDepth and EGL (Epsilon Generation Language), two languages that were introduced earlier in this course. Using the MetaDepth toolbar in AToMPM I can generate a MetaDepth description of both the metamodel (the abstract syntax) and the model (as designed by the modeller). Then, using EGL and both these MetaDepth models, I can compile the model into a Python script that will allow simulation.

11.1. The EGL script

The EGL script that I used to compile the model into Python is rather long and every line might not be clear at a first glance, so in this section I will describe what I have done by listing the line numbers and then explaining what happens behind the numbers.

- 1-4 Get all necessary imports
- 5-12 Create variables for: the total amount of steps the simulation has to run and set the name of the block (in AToMPM) that shows the current step
- 14-36 Write all the necessary functions that do not need to be adapted to the current model (explanation of the content of `__init__`, `_run` and `step` will be explained later in the section about simulation)
- 37 This method sets up the CBD like we would manually do in the MoSIS course, in order to work with the simulator
- 40-49 The most outer loop will find all CBD's in the model. Because multiple CBD's can be created, it is important to find the main, outer CBD. These lines check every link between object, if the destination of the link is the CBD, this means that it is a child of another CBD (property of a containment relationship) an can therefor not be the main CBD.
- 50-133 Handles all CBD's that are not the main CBD.
  - 51-62 A CBD that is not main needs to have input ports and output ports if it wants to receive input or give output respectively. To check if there are any, all links of type `InputConnection` and `OutputConnection` are looped, if the destination of the link is the current CBD, the source of the link is a port associated with this CBD.
  - 64 Creates the lines that initializes the CBD
  - 65 Adds the newly created CBD to the main CBD
  - 67-84 Using all links of type `ContainsBlock`, this loop will add every block that is in the current CBD to the CBD. Blocks that need special attention (because they have more parameters) are `ConstantBlock`, `GenericBlock` and `LoggingBlock`, which is why they get their own case.

- 85-115 After the block is added to the CBD, the connections between this block and other blocks of the CBD have to be added. Because the simulator has its own sequence when adding connections, I forced the right connection to the `input_port_name` because for some blocks the order of addition is important. Lines 87 to 102 handle all the special cases, after which in lines 106 to 113 all the standard IN connections are added (the order no longer matters here).
- 116-131 The cases above do not cover connection coming from the InputPortBlock, so these lines connect all InputPortBlocks to the right blocks that are part of the CBD.

136-239 These lines do pretty much the same as lines 50 to 117 that were discussed in detail above, but add to the main CBD instead of a child CBD. There are a few additions however:

- 179-243 (except 214-225) The main CBD needs to be able to communicate with its child CBD's and for this it uses the `InputPortBlock` and the `OutputPortBlock`, so the addition of these blocks get their own special cases.

247 Concludes the model generation and calls the `_run` method to either start or setup the simulation, depending on the way of execution (more about this later).

247-268 The code needs to be able to give back the results of the simulation. The method created in these lines does just that. A list of tuples is generated, one tuple for each block (that contains an output value), which contains the name of the block (the ID of which is used in AToMPM to find the correct block to modify) and a list with values of this block, the last one in this list being the most recent (of the last simulation step).

272-279 Concludes the file by adding a main method. Calling this method performs a full simulation (not step by step) and returns the results.

12. Simulation

Performing the simulation was by far the most challenging part of this project. I have gone through a number of iterations, each one allowing me to

get somewhat familiar with the method that I will be needing for the next until the project was done. I will start by introducing the simulation toolbar and then go into more detail about each iteration, explaining what I have done and what the challenges were.

12.1. The toolbar

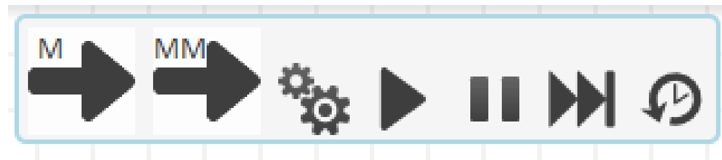

Figure 7: The simulation toolbar

The toolbar used for simulating the model is shown in Figure 7. From left to right the buttons perform following actions:

- Export the current model (a complete CBD model) to metadepth
  the exported file should be named "exported"
- Export the CBD metamodel (CBD.model) to metadepth
  the exported file should be named "CBD"

  note: exported files will be placed in the */mt* folder, not the */exported_to_md* folder
- Automatically compile the MetaDepth files to a Python file for simulation (does what was described in section 11). This button only works on Windows systems since it calls a Windows Batch file. On other systems the compilation has to be done manually.
- Start simulating the model.
- Pause the simulation.
- Perform one big step.
- Reset the simulation to begin values.

12.2. Iteration 1: running the simulation

The first challenge was to actually get the python simulation running by clicking the *simulate* button on the toolbar. At this stage, nothing visual would happen in AToMPM because the result of the simulation was not yet being passed on to AToMPM. I had to base myself on an existing implementation, used for modelling Parallel Devs. Having spent some time figuring out how the calls between AToMPM and the python backend work, I was able to modify the existing Parallel Devs implementation to call my own CBD method and receive back the results. Visually the functionality of the simulator can be seen in Figure 8.

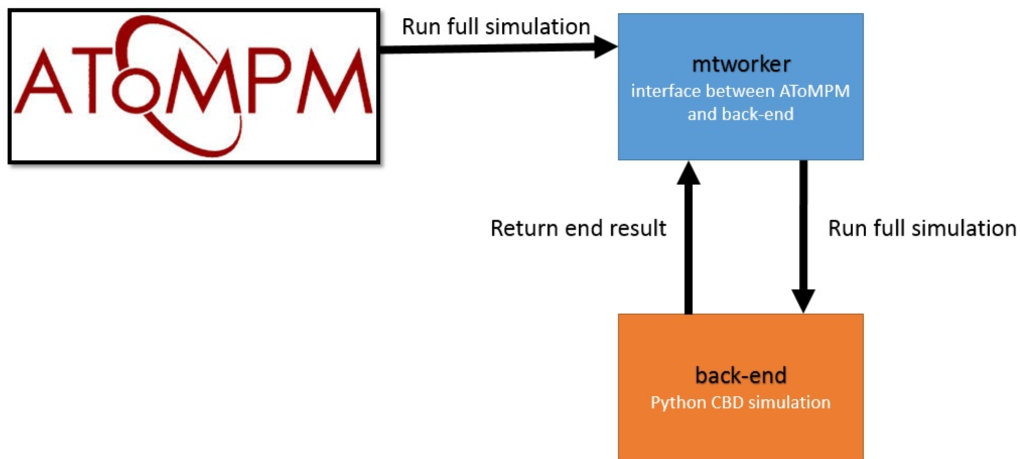

Figure 8: Running the simulation

For the toolbar I adopted the simulate button from the Parallel Devs simulator and set it to call the `cbdsimulate` "method" instead of the `pdevssimulate` "method". A click on this button is handled in the *mtworker* file. Here I imported my (previously) compiled CBD model and ran the simulation, passing the results on the the `_ptcall.simulateCBD()` method. This was not clean at all because *mtworker* should really be spared from any logic, but at that time it did not really matter yet. At this time the `simulateCBD()` method in *cbd.py* (a modified version of the existing *pdevs.py* file) did not do anything with the results.

12.3. Iteration 2: iteration 1 + updating the AToMPM model

Being able to simulate a model without getting to see the results is obviously not very useful. The next logical step in developing a simulator was to make updates to the AToMPM model using the results of the simulation. I can honestly say that this was the most frustrating and most difficult part to get working. Not because the actions are particularly difficult, but I just could not figure out what to call, where to call it and when to call it. I had the solution for Parallel Devs but this was very complex and did not use my simplistic simulation (it already used state charts, see next iteration). It took a lot of testing, printing and tracing but I did get it working eventually, although some of the things I did were not particularly clean or good, but most of these would be fixed while making the next iteration. A visual representation of the second iteration of the simulator can be seen in Figure 9. Not much has changed here, only the returning arrow from *mtworker* to AToMPM has been added.

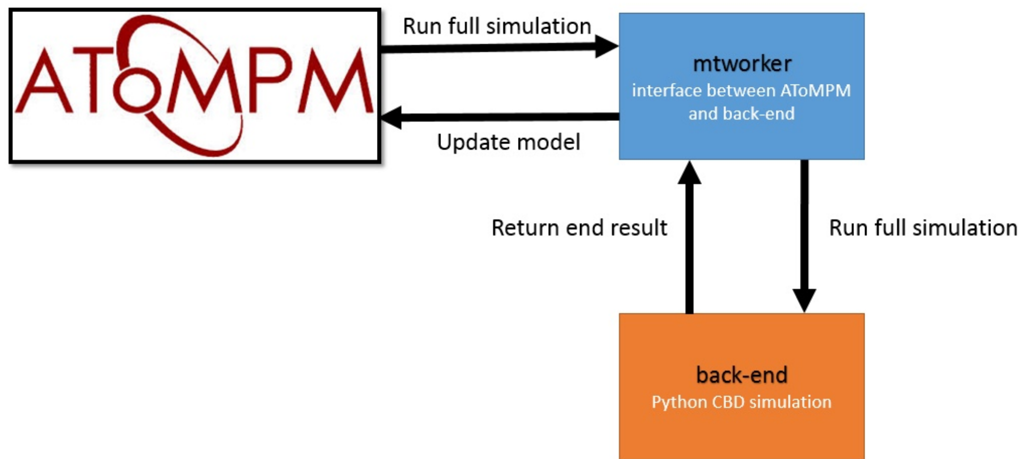

Figure 9: Updating the AToMPM model

I moved the simulation call to the `simulateCBD` method in the *cbd.py* file since *mtworker* should stay free of logic. The results of the simulation are stored in a variable so I could use them wherever I wanted. By analyzing the workings of the Parallel Devs simulator I was able to call successfully call the `utils.httpReq()` method

```
1 utils.httpReq('POST', '127.0.0.1:8124',
```

```
2        '/batchEdit?wid=' + self._aswCommTools['wid'],
3        [self._changeAttr(name, '', values[-1])])
```

(where `name` and `values` are read from the results of the simulation) for each of the blocks, which signals AToMPM to update them with the given value. The only way I could get this to work (although the reason is still unknown to me) was by calling the `_simulate()` method, which in turn called the `_handleStateChange()` method. At this time I did not know what both of these functions really did but since the Parallel Devs simulator used them I did too.

At this time all I had really done was figuring out the interaction between (a): AToMPM and *mtworker* and (b): *mtworker* and the back end. The foundation for the desired and actually correct way of simulating will be researched and tested in the next iteration, while all the remaining iterations build on this foundation to create a fully working simulator.

12.4. Iteration 3: iteration 2 + Using Statecharts for simulation

Terminology: front-end = AToMPM, back-end = the existing python CBD simulator, connection layer = `mtworker` and `ptcall`.

First of all it is important to know that in this iteration I am not yet de/reconstructing the simulator. All I did was learn how to design and compile the xml Statechart implementation and how it interacts with the connection layer. I have had to spend enough time with this part to justify it being a full iteration.
The idea for this iteration was to design a Statechart with two states: *idle* and *finished*. In the transition between these two states, the entire simulation would happen and the results would be sent back to the connection layer so this could in turn update the front-end. Since I again had an example file designed for Parallel Devs, I started by examining this file. With everything I could find in this file, I designed my own Statechart with the two states I needed. The *idle* state would be the initial state and with an external event called *start* a transition is called that (a): makes *finished* the active state, (b): runs the simulation on the back-end, this is still the entire simulation without any control of the user, (c): sends the results of the simulation to the output of the Statechart so that in the connection layer these results can be passed on the the front-end, like in the previous iteration. The Statechart

layer has been added to the image and the result can be seen in Figure 10, all green objects make up the Statechart.

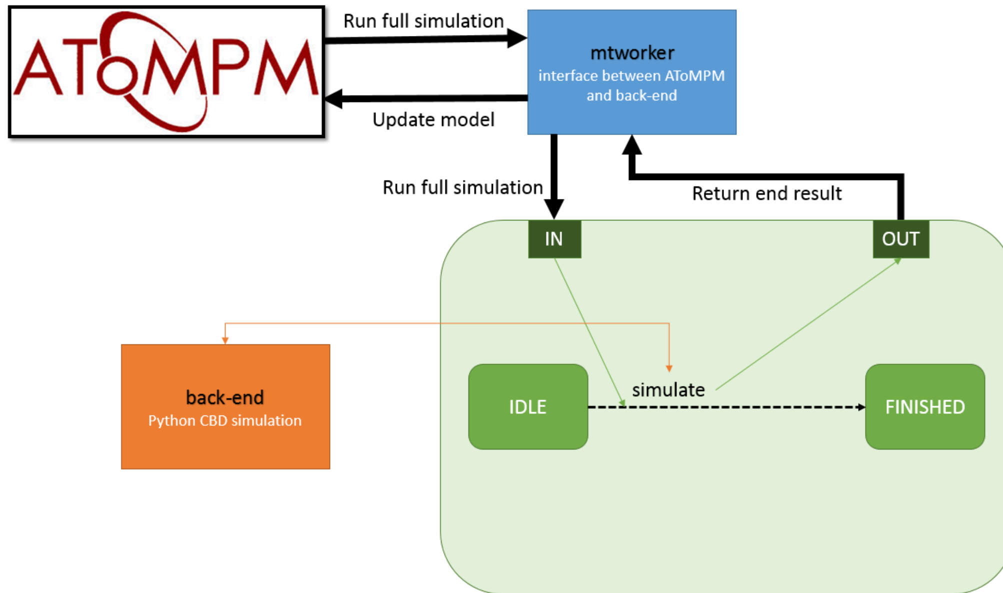

Figure 10: First Statechart

Whilst designing the Statechart in the xml file posed no real problems, getting the compiled version to run on the connection layer did. The compiled Python file uses a controller to interact between the Statechart and the connection layer. When compiled, the file contains a Controller that implements the *EventLoopControllerBase* class:

```
1 class Controller(EventLoopControllerBase):
2   def __init__(self, event_loop_callbacks, finished_callback =
      None):
3     if finished_callback == None: finished_callback = None
4     EventLoopControllerBase.__init__(self, ObjectManager(self),
        event_loop_callbacks, finished_callback)
5     self.addInputPort("request")
6     self.addOutputPort("reply")
7     self.object_manager.createInstance("CBDSimulator", [])
```

This did not seem to work however, and I could not figure out why not, with the research I did I believe it is connected too much with the DEVS model. I looked into the xml to python compiler but could not find a fast fix and I did

not feel like spending too much time trying to figure this out. The solution I eventually found was to manually replace the code above with this code:

```
1 class Controller(ControllerBase):
2   def __init__(self, event_loop_callbacks, finished_callback =
      None):
3     if finished_callback == None: finished_callback = None
4     ControllerBase.__init__(self, ObjectManager(self))
5     self.addInputPort("request")
6     self.addOutputPort("reply")
7     self.object_manager.createInstance("CBDSimulator", [])
```

There is one problem with this: somehow when using the `ControllerBase` class instead of the more extensive `EventLoopControllerBase` class, some automated functionality was lost. Digging through the *statecharts_core* file and following a function call trace, I found a solution: in the `_simulateCBD()` method, which polls the Statechart for changes, I have added two lines:

```
1 self.controller.handleInput(0.0)
2 self.controller.object_manager.stepAll(0.0)
```

This fixes the missing functionality of using the base controller. It might not be the most perfect solution and if I had more time to complete this assignment (a lot of time already had already gone into this project at this time and a lot still had to be done) I might have found a better solution, but for now it works.

Setting up the interaction between the connection layer and the Statechart was pretty straightforward. I could use the calls of the example ParallelDevs model to both send Events to the Statechart as well as poll the reply ports without changing much (the lines discussed in the previous paragraph and adapting the `addInput` calls to the right type of event).

12.5. Iteration 4: iteration 3 + de/reconstruction of the simulator to/from Statechart

This iteration meant implementing changes in every layer of the project. Like all previous iterations I will first discuss the global goal and then go into more detail on all changes.

In this iteration I will take out part of the existing simulator and model this using the Statecharts. What I want to do is enable step by step execution. To do this, one state named *working* and a couple of transitions are

added to the previous Statechart, as shown in Figure 11. The working state has a transition to itself that can be executed by an external event ("step") as long as the end condition (the current time step being smaller than the total amount of time steps of the simulation) is not met. In this transition, one step is executed in the simulation.

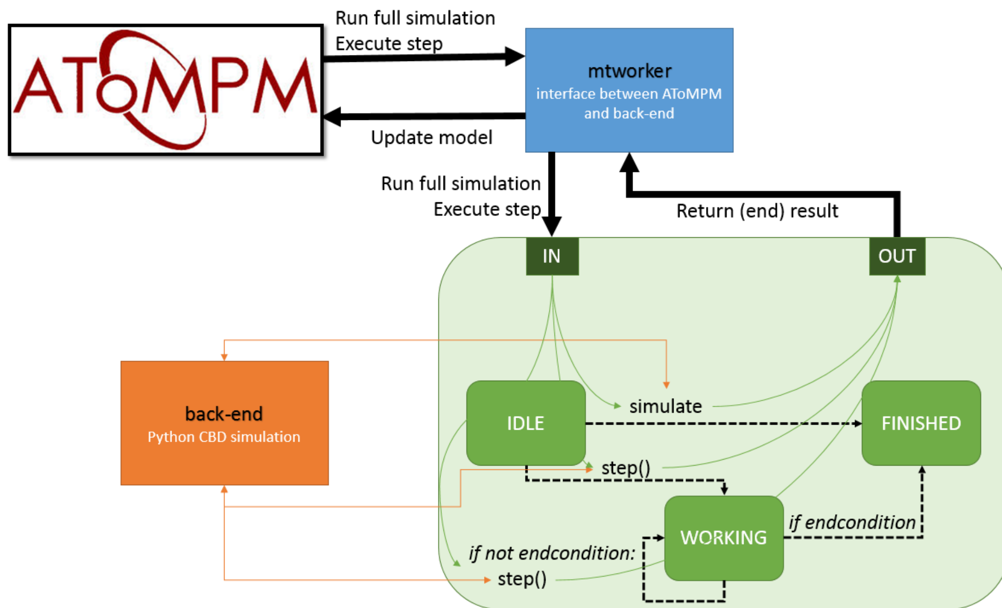

Figure 11: De/reconstruction

Like I said before, to make this work a lot of changes had to be made to a lot of files. All files that I have discussed in previous chapters already have these changes implemented so here I will discuss what I have done to each file:

- **The AToMPM model:** The changes here were not really that exciting. I added the CurrentTimeStep class so the user could easily see how many steps he had already performed, this was previously not necessary because all the steps would always be executed when simulating.
- **The CBD simulator:** This would be the first time I really modified the structure of the CBD simulator, which was up until now still roughly the same as the one we developed during the MoSIS course.

Two methods were added: `setup()` and `step()`. Previously the simulation would be executed by the `run()` setup, that got the number of time steps as a parameter and ran the `__step()` method as often as requested. This does not allow the modeller to have any gradual control. The first new method (`setup()`) is only called once and prepares the CBD to be executed. The second new method(`step()`) performs all the actions that would previously be done automatically by the `run()` method.

- **The MetaDepth to Python exporter**: Just running the main method of the compiled python file to start the simulation no longer sufficed. Somehow the `Exported` class needed to be able to differentiate its functionality on wether there a full simulation is requested or a step by step simulation, because at this time both were possible. Following changes were made:

  1. A new variable `CUR_TIME_STEP_NAME` was added, that holds the AToMPM id of the the class that shows the current time step.
  2. The CBD name can be passed to the constructor instead of being constant.
  3. This allows the `_run()` method to differentiate between a full simulation (where it executes the entire simulation like it always used to) and a step by step simulation, where it only calls the new `setup()` method, performing no steps or simulation in the process.
  4. A `step()` method was added that calls the `step()` method in the CBD simulator, as discussed above.
  5. The construction of the result list no longer happens immediately after the `_run()` method had been called. This used to be the case because after this call all the steps and calculations would have been executed and the full list of results could be returned. Now after this call there is a chance that no steps have been executed yet and that no results have been generated. Now retrieving the results from the CBD simulator happens in a new method called `getResults()`, which can be called at any point before, during or after the simulation.

- **The simulation toolbar:** received a new button that performs a big step and calls the appropriate method in the connection layer.

- **The connection layer:** Here a new method was created (`bigstep()`), which inputs a new type of event to the Statechart, named "step".

At this point, the modeller could either simulate his model using step by step execution or by requesting a full simulation. These two methods had almost nothing in common, which also meant that they were not compatible: it was not possible to continue running automatically when a number of manual big steps were already executed. The next iteration takes care of this problem.

12.6. Iteration 5: iteration 4 + eliminating full simulation

There is one solution to the problem that I introduced above: using automatic step by step simulation to "fake" a complete simulation. This was a rather small modification. Figure 12 shows the modified state chart.

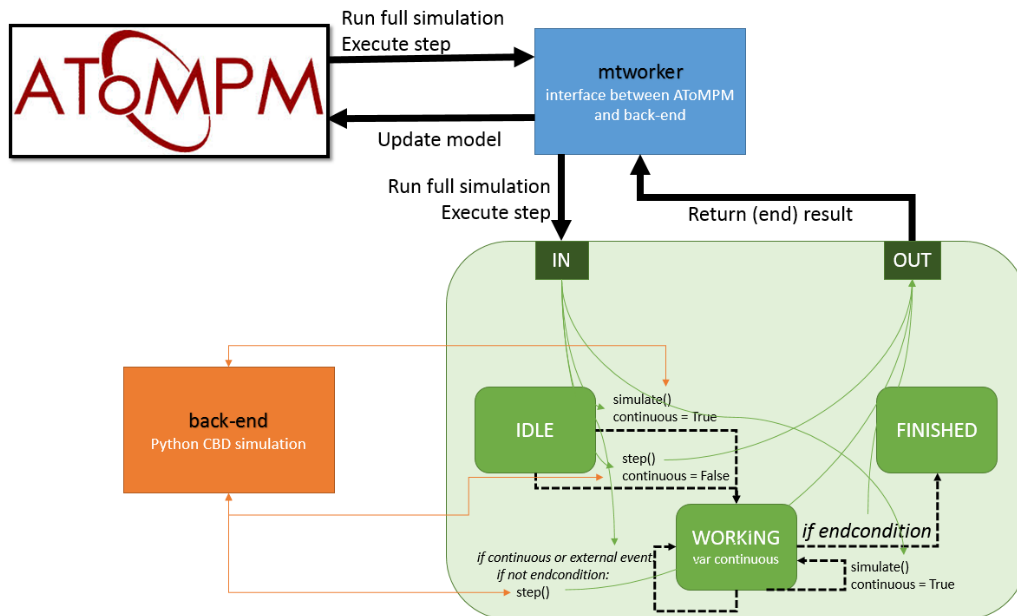

Figure 12: Eliminating full simulation

As you can see the transition from the *idle* state to the *finished* state, previously used for full simulations, has been replaced by a transition from the *idle* state to the *working* state. Apart from this, I introduces a variable named *continuous*, which indicates whether we are doing a continuous (full)

simulation (`True`) or a manual step by step simulation (`False`). The transition from *working* to itself has also gotten an upgrade: it now executes when (a): an external *step* event is received or (b): if the *continuous* variable is set to `True`. When the modeller wants a full simulation, *continuous* will be set to `True` and the transition will keep firing until the end condition is met, thus acting like it is running a complete situation.

These steps make all functionality in other files (the CBD simulator and the from MetaDepth exported file) unnecessary, but I will kept them in because they do not mean any harm and they may become useful in future work or testing.

The other (new) transition from *working* to itself allows the modeller to continue the simulation in automatic mode after first having performed some steps manually.

12.7. Iteration 6: iteration 5 + reset

It would be pretty handy if the modeller could actually perform his simulation multiple times without having to reload everything. This small iteration adds all functionality to it. In Figure 13 you can see the new transition from *working* and *finished* to *idle*. By adding a minimal amount of options to the connection layer (sending an `addInput()` call to the Statechart when the "reset" button is pressed) the model can be reset at any time. When entering the *idle* state, the CBD is re-initialized and loses all its previous values, so the simulation can be started over.

12.8. Iteration 7: iteration 6 + pausing the simulation

The last iteration seemed to be easy at first but still required quite a bit of effort to get it working. I need to be able to pause the simulation. To do this I added one last state to the Statechart, named *paused*. When the user presses the pause button, the (currently continuously running) Statechart would transition into the paused state and stop executing. Immediately I noticed that this would not be as easy as it seems. Figure 14 shows the entire simulation engine including the new paused state.

The first problem I encountered was that the Statechart simulation is actually running instantly, whilst the AToMPM model shows a slowed down version of the simulation to actually make it possible to see what is happening. This means that when I press the pause button while the simulation

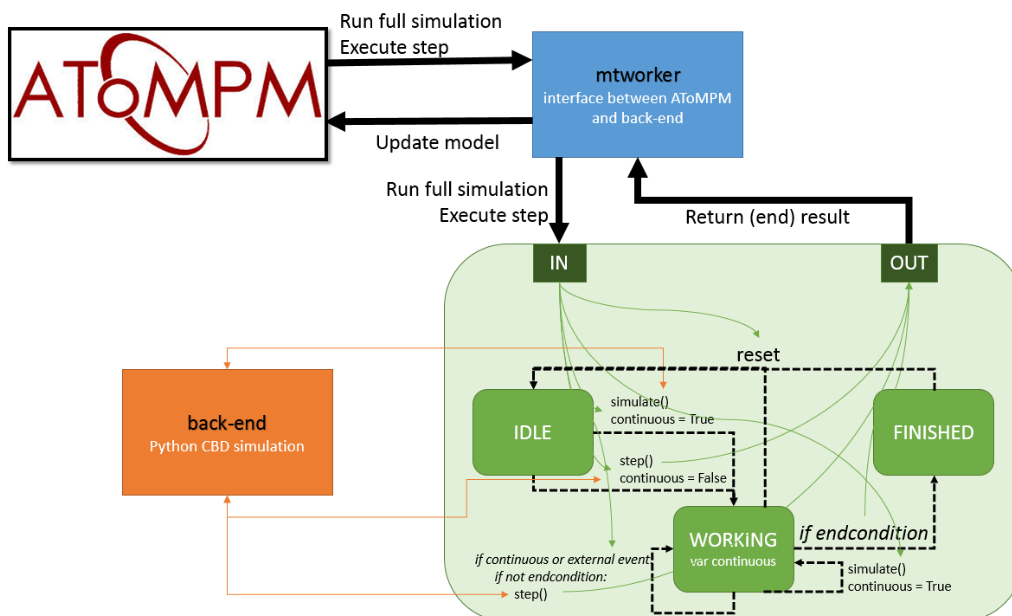

Figure 13: Enable resetting of the simulation

seems to be halfway done, the calculation of the simulation of the Statechart is already complete, so pausing did not do anything.

The solution to this was just sleeping in the simulation for a time longer than the update delay in AToMPM. Or so I thought. I read my way through the file responsible for input handling (*statecharts_core.py*) and after a while I found the reason it would not work: the simulation engine gives priority to internal transition (that happen automatically), so the pause event did not get handled unless no internal or automatic transitions (like the ones that cause the automatic stepping) remain. So even if I did sleep the simulation nothing would change if I pressed the pause button, it would just take a lot longer to execute. No external transition could force itself between all continuous series of internal ones.

Now that I had found my problem, I could try and solve it. The only way I could think of is to set the `continuous` variable to `False` so the internal transition just would not fire. Of course the simulation still needed to be automatic. The fix for this was to start a new thread which would sleep for a

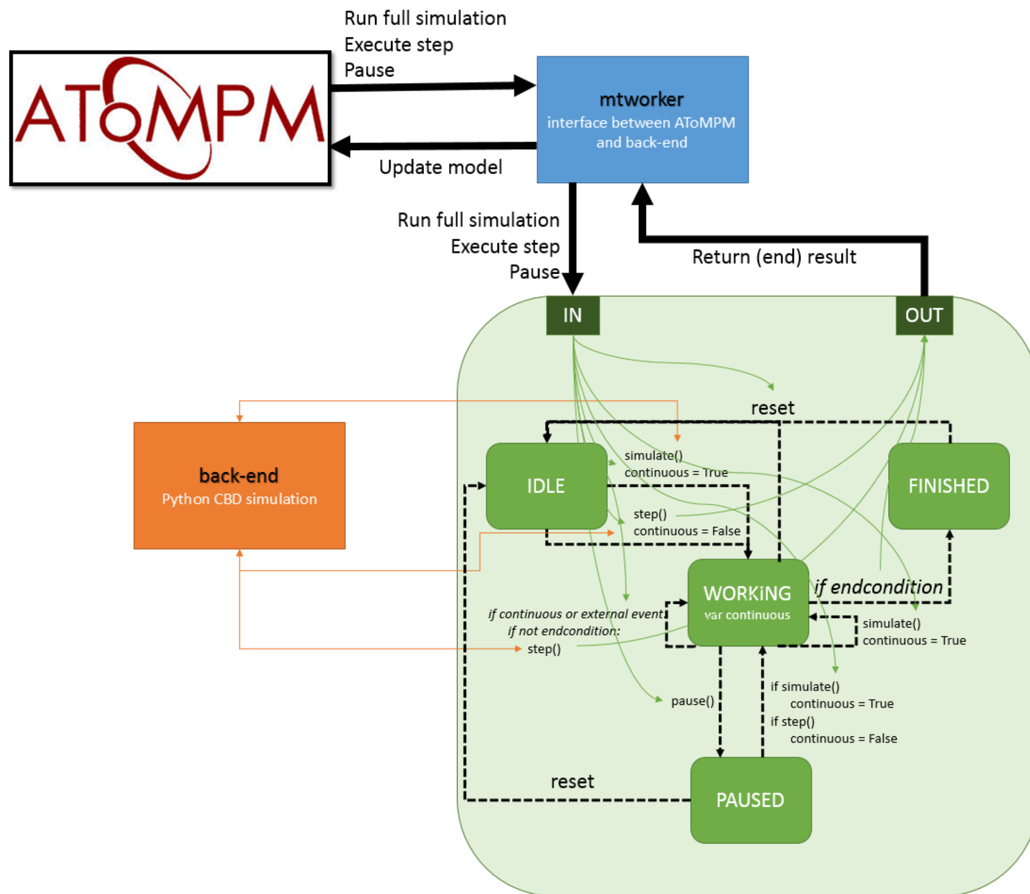

Figure 14: Pausing the simulation

certain time (set to 0.25 seconds) before resetting the `continuous` variable to `True`. I expected that it would now work... but it still did not, the simulation would not continue automatically after having reset the variable. Time to find and solve the next problem.

I found out that the Statechart simulation works as follows:

1. Wait for external input event
2. Receive input event
3. Perform transition on internal events
4. Check if any internal transitions fire, if so, perform transition and repeat this step

5. Go back to step 1

What does this mean for my solution? Because I set the `continuous` variable to `False`, step 4 would not fire and I would go back to waiting for an external input event, which would of course not come because the user is assuming the simulation is running automatically. The (rather ugly, but the only one I could think of) solution was to generate an external transition from within the new thread, forcing the simulation to continue after sleeping.

After all these fixes, pausing the simulation finally works and the functionality for this project is complete.

13. Future work

It is fair to say that this project is not perfect, for this to be the case a lot more time has to be invested. During the development of this project I have cut some corners and done things of lesser importance imperfectly to maximize the global usefulness of the application. This section will give a short overview of what might be still be done and how I would envision doing it if I were to continue working on this.

13.1. The AToMPM syntax

Both visually and usability-wise some improvements can be made to the model:

13.1.1. Constraints

Other than having a particular number of inputs on each block, no constraints (global or block-wise) are implemented at this time, because they were not essential to the building and simulation of CBD's. In fact, if the modeller does everything he should and does not try to make connections that are not logical, no constraints should be necessary. Following constraints could be imlemented:

- There can only be exactly one main CBD, other CBD's have to be children of this main CBD.
- There has to be exactly one *NumTimeSteps* instance.
- There has to be exactly one *CurrentTimeStep* instance.

- Blocks with no output ports should not be able to connect to another block (e.g. the *LoggingBlock*).
- When using a child CBD, block from the parent CBD should always connect to blocks of the child CBD through *InputPortBlock*s or *OutputPortBlock*s, never directly.
- Blocks should always be contained in a CBD.

13.1.2. Visual Improvements

- *InputPortBlock*s and *OutputPortBlock*s should snap to their respective CBD when they are connected. This seems a simple fix but since multiple input ports and multiple output ports can be added, I could not just say snap to that target, I would have to build in a lot more logic in order for it to work so I decided I could better spend my time on other, more important things.
- A lot can be done on the visual appearance of the blocks and their values, just like the previous point, the current implementation is functional and more important things could be done in stead.
- It could be possible to also snap incoming and outgoing connections to block to the visually provided ports, however because links are made by clicking, I found it to be almost just as clean to just connect it there in the first place.

13.2. Exporting MetaDepth to Python

For the current functionality the exporter works just like it has to and needs no further modification. If someone was to add or modify functionality in the future, he would of course also have to adapt the exporter.

One very small improvement was to enable the compile button to work on all systems, instead of only on Windows systems.

13.3. The connection layer between AToMPM and the Statecharts (mtworker, ptcal, ...)

Just like the previous section, this works as it should at this time, but any extra or modified functionality may need adaptations in these files.

13.4. The CBD back-end simulator

In order to further develop the level of detail the user can debug with, following things could be added:

- Small step: allowing the user to do one block calculation instead only whole CBD step simulations. This could be done by pulling apart the `__computeBlocks()` method of the simulator, where every component (block) of the CBD is calculated individually. By removing the automatic for loop and replacing this with manual steps the same functionality should be replicable.
- Backward Big step: allowing the modeller the go back a time step might give him the possibility to debug faster by repeating a couple of steps without having to reset the simulation constantly. It might be as easy as calling the `__step()` method on with the previous `curIteration` value, but I have not tested this.
- Backward Small step: the same as backward big step but with block level steps.
- Breakpoints: Although I do not have any idea whether this will work, implementing a special type of block that notifies the simulator that it needs to stop at this time might do the job. I have no idea about how this would work at this time.

Adding any of these options will automatically lead to changes in other major parts of the program.

14. Important files

This section provides an overview of all files that were modified or created for this project. Each folder gets its own subsection.

14.1. ./atompm/

- **AToMPM.bat** Starts up AToMPM, for ease of use
- **CBDServer.bat** Starts up the server, for ease of use
- **compile_metadepth.bat** Used by the toolbar to automatically compile the MetaDepth models into Python

14.2. ./atompm/users/michael2/Formalisms/CBDs/

- **/block_images** Contains all the images for the concrete syntax model
- **CBD.defaultIcons.model** The concrete syntax model for the CBD formalism
- **CBD.defaultIcons.metamodel** The CBD formalism toolbar, used for creating user models
- **CBD.model** The UML abstract syntax model for the CBD formalism
- **CBD.metamodel** The compiled abstract syntax model
- **Demo.model** An example model which demonstrates the use of the formalism

14.3. ./atompm/users/michael2/Toolbars/CBDs/

- *icons* Images used for the simulation toolbar
- **Simulation.buttons.model** The CBD simulation toolbar that allows the exporting, compilation and simulation of a CBD model

14.4. ./atompm/mt/

- **CBD.mdepth** The MetaDepth file of the metamodel
- **CBD.py** The existing (but now modified) simulation file that is used in the back-end
- **exported.mdepth** The MetaDepth file of the loaded model (currently Demo.model)
- **generate_python.egl** The EGL file that is responsible for compiling the two mdepth files above to the project.py Python file
- **mtworker.py** The connection between AToMPM and the simulator, was designed (not by me) for ParallelDevs but is extended and modified to work with CBDs
- **project.py** The Python model, generated from MetaDepth
- This folder also contains some extra python files that were required for the CBD.py simulator to run, but were not modified so they are not listed here.

14.5. ./atompm/mt/ptcal/

- **cbd.py** The connection layer responsible for handling AToMPM request, sending requests to the back-end and updating AToMPM with the results

14.6. ./pypdevs/src/

- **cbd.xml** The XML Statechart model that is responsible for simulating the model

- **cbdsim.py** The compiled version of this Statechart, with the correct controller

- **cbdbad.py** The compiled version of the Statechart without the correct controller (this is the actual result of compilation, but does not work)

- **replace.py** The correct controller. By replacing the controller from cbdbad.py with the controller in this file, you get cbdsim.py

15. Final thoughts

I asked for this project because, in the MoSIS course, I felt that there had to be a better way to design and get comfortable with CBD's than creating and testing them in a text-based way. At first I was thinking of the best way to make the environment but it was obvious that this would have to be AToMPM, we use it for almost every single assignment in this course (Model Driven Engineering) and even some assignments from the MoSIS course, so every student is familiar with it.

During the development of this project I noticed that I used a lot of what this course has taught me, abstract syntax modelling, concrete syntax modelling, code generation, and even simulation (altough this part was much more challenging than the previous assignments). I likes this because it gave me the opportunity to refresh my knowledge on all this subjects and in general the entire course one last time.

Of course this project was not all fun and happy coding. Creating the abstract and concrete syntax was not all too hard, writing the compiler for compiling models from MetaDepth to Python was challenging and took some

hard thinking, but it never posed any real problems. The simulation and debugging, however, was at times a real nightmare. Starting from an existing framework (in this case used for ParallelDevs) has some advantages: a lot of coding is already done, you can see how it is done and try to replicate it, ..., but the disadvantages can be cruel. Because I did not design the framework I did not know the inner workings and this had huge effect on the speed and efficiency I could work with. Since the framework was designed for ParallelDevs, it was not always easy to just redesign it for CBD's without going very deep into the design and figuring out why something does not work, at times realising that it would not work the way I hoped it would without having to spend a lot of time modifying the logic of the framework. At those moments I opted for a less clean, yet more time efficient, solution. If I was to do all the work I did for this project again with the knowledge that I have now, I would be done in less than a third of the time, but those are the glorious qualities of programming (and I do mean that I love these types of challenges).

All in all I am pleased with the results, I learned a lot, I enjoyed working on it and the functionality (model creation, exporting, compiling, full simulation, step by step simulation and pausing, to name the most significant) is in my mind very decent for the time I had to spend on this project.

16. Bibliography

Van Mierlo, S., Van Tendeloo, Y., Barroca, B., Mustafiz, S., Vangheluwe, H., 2014. Explicit modelling of a parallel devs experimentation environment.

Vangheluwe, H., Denil, J., Mustafiz, S., Riegelhaupt, D., Van Mierlo, S., 2014. Explicit modelling of a cbd experimentation environment. In: Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative. DEVS '14. Society for Computer Simulation International, San Diego, CA, USA, pp. 13:1–13:8.
URL http://dl.acm.org/citation.cfm?id=2665008.2665021