Visual modelling environment for CBD's

part 1: Reading assignment

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, yet there is no decent or modern visual editor that also provides running a simulation and debugging. I set out to design a modelling environment in AToMPM that does all of these things. This paper contains the results of research that has been done on the subject of debugging CBD's and AToMPM.

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

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 , and , ...). A collection of blocks is connection through signals, which connect the output of one block to the input of another. Most blocks have a particular amount 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 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

[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.

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 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

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

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.

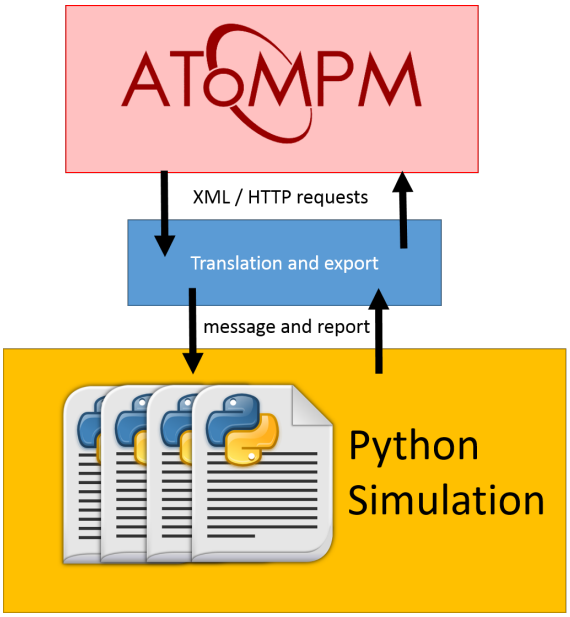

Figure 1: Simplified CBD debugging architecture

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

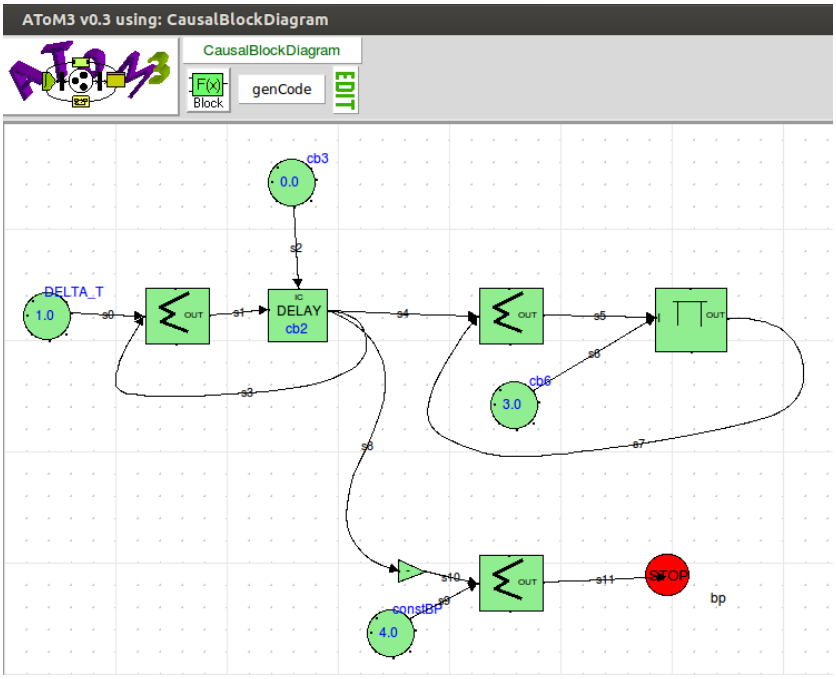

Figure 2: A CBD with a breakpoint block (red) modelled in the depricated AToM3 environment (Vangheluwe et al., 2014)

to be modified in order to work with this new type of block. The specific 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.

9. 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`