Causal Block Diagrams: Compiler to LaTeX and DEVS

Nicolas Demarbaix
University of Antwerp
Antwerp, Belgium
nicolas.demarbaix@student.uantwerpen.be

Abstract

In this report I present the results of my project for the course Model Driven Engineering. This project consists of developing a Causal Block Diagram (CBD) to LaTeX and DEVS compiler. Using this compiler one can generate text based documents (in LaTeX) that give a detailed description of the CBD model on one hand. On the other hand, one can transform the CBD model into an equivalent DEVS model. This will allow us to use, for example, the PypDevs simulation framework for DEVS to simulate the Causal Block Diagram in a different environment and compare the results. Also it allows us to use the generated CBD AtomicDEVS as a component in another DEVS model. This way it is for example relatively easy to link different CBD's together in an arbitrary order.

Keywords: Model Driven Engineering, Causal Block Diagrams, Compiler, LaTeX, DEVS

1. Introduction

During the course *Model Driven Engineering* many different aspects of Modelling were discussed. Topics such as Meta-Modelling, Concrete Visual Syntax, Semantics and Transformations were brought to attention. In *Modelling of Software-Intensive Systems*, a course coherent to *Model Driven Engineering*, different Modelling Formalism were presented. Using both the theoretical and practical aspects of both courses, a Causal Block Diagram (CBD) to LaTeX and DEVS compiler will be developed in the light of a final project for the course *Model Driven Engineering*.

Both aspects of this compiler were originally planned to be developed using the AtomPM modelling tool. However due to issues with attempting to transform a CBD model in AToMPM to both a *LaTeX* and *DEVS* model, I decided to leave this path and implemented the compiler using Python[1].

1.1. Motivation

The reason(s) for developing a CBD to LaTeX/DEVS compiler are the following. To start let us take a look at the CBD to LaTeX part of this compiler. CBD models are a good way to model for example a process or a set of equations. It is however not always easy to understand the model by simply looking at it. A textual description of such a CBD model could overcome this burden. By providing a CBD to LaTeX compiler, one could easily retrieve information about the CBD such as block definitions, interconnectivity of blocks, possible algebraic loops in the model, etcetera.

One of the reasons why one might want to transform a CBD model into a DEVS model is the ease of using events in DEVS. Say for example that you want to study the effect of a certain input value to an entire component of the CBD. This can easily be achieved in the DEVS formalism by using external transitions and event generation. Another advantage of transforming CBD into DEVS is that one could verify the model by exporting the DEVS model to PypDevs.

1.2. Concept

The general concept of the CBD compiler is the following. The '*to LaTeX*' aspect of the Compiler is to provide a textual representation of the CBD model. In this textual representation details such as block structure, connections between blocks and functions/values of the blocks can be included. Moreover, by using an intermediate language[2], the algebraic loops that could occur in a CBD model might already be solved and a description of these loops could also be added to the textual representation.

[1] Issues included for example AToMPM crashes when exporting a transformed dependency graph to metaDepth and not being able to add a method to the DEVS output file for calculating the algebraic loops. I will include my work in AToMPM with this project. It includes constructing a CBD model, transforming this model to a Dependency Graph model and applying the dependency graph algorithms to constructu the strong components.

[2] More information about the concept of this intermediate language can be found in Sections 2 and 3.

On the other hand we have the '*to DEVS*' aspect of the compiler. This aspect provides the means to construct a DEVS model that is directly related to the CBD model. The resulting DEVS model will consist of all different elements of the DEVS formalism[3]. Using these elements the main concept of this aspect will be to create an Atomic DEVS element that contains a state for each strong component. Inside such a state, the possible cyclic dependencies in such a component can be resolved to find a solution for the algebraic loop (see later). The general idea in this concept is that the AtomicDEVS that represents the Causal Block diagram can act as a single component, which can be used by itself or inside a network of AtomicDEVS Components.

1.3. Structure

The structure of this reports is as follows. In Section 2, a theoretical basis for this project is provided. In Section 3 I will discuss my design intentions for the compiler itself. Next, in Section 4 I will discuss the actual implementation of both the CBD to LaTeX as the CBD to DEVS compiler. In Section 5 I will discuss some general results about the implementation. Lastly, in Section 6 I conclude on the project and will present some possible Future Work.

2. Theoretical Background

In this section I will give a brief overview of the theoretical background on which this project is based. I will assume a decent knowledge about these topics such that I do not need to go in too much detail, which would be outside the scope of this report. I will start by presenting the constructs that will be used to build an intermediate language (see Section 3). Next I will present the mapping between the CBD model and the DEVS model without going into details about design intentions. Next I will discuss the issue of algebraic loops and how they will be solved. Lastly I will take a quick look at an algorithm for dynamically changing the rate of a block inside a CBD model.

2.1. Intermediate Language Constructs

The intermediate language that will be used when transforming CBD models to either DEVS or LaTeX will consist of the dependency graph of the Causal Block Diagram along with certain dictionaries (see Section 4) that will store certain aspects of the CBD. This means that we need to provide the proper algorithms to transform the source CBD model into such a dependency graph. Of course, the methods to construct the dependency graph are already provided in the CBD implementation[4]. Why this structure is used as intermediate language is discussed in Section 3.

The algorithms that are used to construct the dependency graph can be found in Appendix A. These algorithms come directly from the course page for *Modelling of Software-Intensive Systems*[5]. As stated, the actual implementation of these algorithms originates from a project for the course Modelling of Software-Intesive Systems. We can use the dependency graph that is constructed using these algorithms to determine the order in which blocks should be computed, and more importantly, to discover whether the causal block diagram contains loops that can be solved.

It is important to note that the exact ordering of the nodes will be random as the "root" node for the algorithm is choosen at random. This is of course the case since we are working with cyclic graphs which contain no single root node. However this random ordering will only occur in the Topological Sort algorithm as the algorithm for finding the strong components of the dependency graph should always return the same set of strong components.

Using the above algorithms, a depency graph is built from the CBD. Using both the CBD model itself as the dependency graph, we will construct the following dictionaries, which are also part of this intermediate language:

1. A "depends on" dictionary containing per block all the blocks it depends on
2. A "influences" dictionary containing per block all the blocks it influences
3. A "component" dictionary containing per strong component in the dependency graph all blocks that belong to this component
4. A "types" dictionary containing per block its block type

Why we need such dictionaries and where we will use them will become clear in the next sections.

2.2. CBD to DEVS mapping

As defined by Zeigler et al. (2000), an Atomic DEVS model M can be written as $M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$ where

1. X — Input Set
2. S — State Set
3. Y — Output Set
4. $\delta_{int} : S \rightarrow S$ — Internal Transition
5. $\delta_{ext} : QxX \rightarrow S$ — External Transition
6. $Q = \{(s, e)| s \in S, 0 \leq e \leq ta(s)\}$ — total state, e is elapsed time

[3]The elements of the formalism that are refered to are *Atomic DEVS*, *Coupled DEVS*, *Events*, *Internal Transitions*, *External Transitions* and *State Definitions*

[4]The implementation for these methods was constructed during the course Modeling of Software Intensive Systems

[5]http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/CBD/topsort.pdf
http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/CBD/strongcomp.pdf

7. $ta : S \to R^+_{0,\text{inf}}$ time advance function
8. $\lambda : S \to Y$ output function

Using this definition of Atomic DEVS we will now take a look at how the CBD model will be mapped onto such a DEVS. Note that each strong component will be mapped onto a single state. Each state of the Atomic DEVS will thus either contain:

- A single element. In this case the component in the dependency graph of the Causal Block Diagram was not cyclic and contained only a single block
- A number of elements. In this case the component was cyclic.

Actually stating that the state in the Atomic DEVS "contains" a single or a number of elements is not entirely correct. Each state will contain all the values of the blocks in the CBD. In each state, the values of the blocks belonging to the component to which the state corresponds will be computed and updated (more on this topic in Section 4.
For each Atomic DEVS that is generated by the transformation we see that $X = Y = \{Signal_V alues\}$ where SignalValues is a dictionary that is returned by the internal transition going from this state to the next. SignalValues will contain the updated values for the blocks that were computed in this state.
The state space S of the Atomic Devs will consist of all the states that are generated for the different components in the dependency graph. But not for each component a state will be generated. For those components where we can determine the block values at setup time (by propagating the block values along the CBD, for as far as possible), there will be no state in the Atomic DEVS.
The internal transitions of the Atomic DEVS will corresponds to links between the strong components in the dependency graph of the Causal Block Diagram. More specifically, the strong components will have a certain ordering in the dependency graph (based on the dependencies of the blocks belonging to these components). Each link between two states will consist of an internal transition between from the state of $component_i$ to the state of $component_{i+1}$.
There will be no explicit external transitions present in the Atomic DEVS model. In case the AtomicDEVS model serves as a component in a network of AtomicDEVS models, external transitions and the corresponding ports will need to be added, but this can be performed quite easy by extending the method "extTransition" in the class CBD.
The total state of the Atomic DEVS will consist of the output signal of each block represented by a value in the Atomic DEVS State, CBDState, at a certain point in time. This output signal will of course be based on the input signals at the same given point in time.
The time advance function for each state will be based on the rate of the corresponding block. The rate corresponds (for example) to the number of steps that are taken each second by the block. The time advance function might change over time if the rate of the corresponding blocks changes dynamically (see Section 2.4).
Lastly, the output function λ does not produce any output for this Atomic DEVS. If output is needed when connecting the Atomic DEVS to other Atomic DEVS, the method "outputFunc" in the class CBD can be adapted to return a certain output set.

2.3. Algebraic Loops

Algebraic loops correspond to cycles in the dependency graph of the CBD model. In terms of CBD models an algebraic loop occurs when the output of a certain block is connected to its own input, either directly or indirectly. The problem with these algebraic loops is that the output result of the corresponding block(s) cannot be computed explicitly. Or interpreted differently, their output signals cannot be determined based on the initial state of the model.

There are many ways to solve such an algebraic loop. To minimize the complexity that corresponds with these loops, we will first try and determine whether the algebraic loop is linear. If so it can be solved using techniques such as *Guassian Elimination*. If it is non-linear, more advanced techniques are required.

To determine the linearity of the algebraic loop, we base ourselves on the definition of linearity and the structure of the blocks and the CBD model. From the Encyclopedia of Math (2011) we include the following definition for linear equations. Given a set of N variables $\{x_i | i \in \{0, \ldots, N-1\}\}$ and a set of known values $\{a_i, b\}$ where $i = 0, \ldots, N-1$ an equation of the form $a_0x_0 + a_1x_1 + a_2x_2 + \cdots + a_{N-1}x_{N-1} = b$ is always linear. Otherwise interpreted we see that an equation is thus linear if all variables in the equation are of first degree. Also the multiplication of two or more unknowns (e.g. $z = xy$) results in an equation being nonlinear.

Based on this definition, I will now present a list of block combinations in a CBD model that can be classified as nonlinear. In case these combinations occur, an error message will be produced and any ongoing procedures will end. In all other cases however, we classify the equations as linear and the procedure can continue. The following combinations of blocks in a CBD model yield non-linear equations:

- The precense of a RootBlock always results in an equation being non-linear, as it computes $\sqrt{x} = x^{1/2}$. We see that the unknown x is not of first degree in this case, so these equantions will be non-linear.
- The abscence of at least one ConstantBlock as input for a ProductBlock. In case none of the inputs of the ProductBlock are of type ConstantBlock, then we see

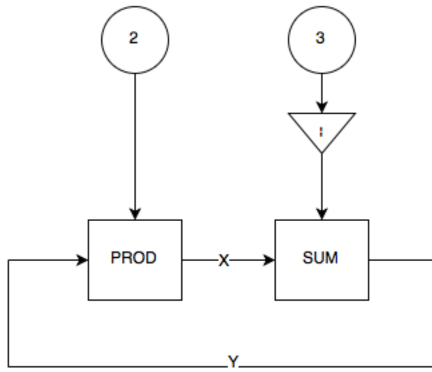


Figure 1: Example of a basic algebraic loop in CBD.

that the output signal 'z' of the ProductBlock is calculated as $z = x*y$. We thus see that this ProductBlock yields a non linear equation.

- An exception to this last rule is that instead of a ConstantBlock, the presence of an Integrator- or DerivatorBlock will also result in a linear equation. This can be explained by the fact that these blocks have an Initial Condition which is valid at the initial state of the model. This Initial Condition must be defined at time zero, therefor the output signal of these blocks will also be known in the initial state. This also counts for a DelayBlock, as its initial condition should be known at the initial state of the model. Furthermore, if at least one of the inputs is a WireBlock, InputPortBlock or OutputPortBlock the resulting equation will also be linear, as these blocks simply pass values from one block to another.

Now let us propose a way of dealing with these algebraic loops in case they are in fact linear[6]. A linear algebraic loop can be written as a system of linear equations. For example, the algebraic loop in Figure 1 can be written as the following system of equations:

$$\begin{cases} y = x - 3 \\ x = 2 * y \end{cases}$$

An algebraic loop can thus be solved by constructing such a system based on the cycle of blocks. Using the set of blocks that form the linear component, a matrix M1 and M2 will be generated. These matrices have sizes $M1_{numvar,numvar}$ and $M2_{numvar}$ where the value numvar is the number of variables in the set of equations, i.e. the number of blocks in the component. These matrices can then be used as input for a *Gaussian Solver* that calculates the resulting value for each block. An example of such matrices is provided in Sections 3 and 4. Using these results the appropriate state variables can be update upon transitioning to a new state.

[6]Remember, if an algebraic loop is non-linear, we decide to stop the execution of the program

2.4. Dynamic Block Rate

In her research internship, Christis (2012-2013) introduced the concept of Adaptive Derivator- and IntegratorBlocks. By monitoring the error between the calculated result (by the CBD) and the analytical solution, the rate (or step size) of these blocks is adapted to ensure that the error can be reasonibly minimized. By calculating two different derivative approximations $f_most_exact(x) = \frac{f(x)-f(x-\delta)}{\delta}$ and $f_less_exact(x) = \frac{f(x)-f(x-2\delta)}{2\delta}$ and calculating the error as $error$ = f_most_exact(x) - f_less_exact(x) one can make an estimate of the deviation of the calculated result from the analytical solution. If this deviation is too large (or too small), the step size δ is adapted such that the error can be minimized.

This theoretical base can provide a good starting point to look at the rate of the blocks during simulation. Inside the class CBD, which is the AtomicDEVS model of the original CBD model, a method **updateRate** is provided. This method is a skeleton where the user can implement a certain check to see whether the results in the Atomic DEVS model are correct. If the deviation, based on this check, is too large, the rate of the CBD will be updated with a certain, user-defined factor.

A possible error check that could be performed is the error monitoring described by Christis (2012-2013). This is however not easy to accomplish in the current state of the compiler, as the CBD is flattened before compilation starts (see Section 4). The output values of the blocks inside Derivator- and IntegratorBlock are computed in separate states. Therefore, to allow such monitoring, one should add a second variable inside the CBDState state definition for each such block and update this accordingly with different values for δ. In the current state of the compiler, it is easier to compare the value of the OutputPortBlock of the Derivator- or IntegratorBlock with the known analytical solution of the input equation.

3. Design

Now that we have provided a theoretical basis for the implementation of this compiler it is time to look at its design. I will start by giving some general design insights, after which I will discuss both the *LaTeX* and *DEVS* aspects on more detail.

3.1. General Design

As mentioned earlier the CBD model will not directly be transformed into a DEVS model/LaTeX file. Rather we will first transform the CBD model to an intermediate language. This intermediate language will be the representation of a Dependency Graph of the CBD model. The reason I choose this design is that we can first solve both the topological sort and strong component algorithm in

this Dependency Graph. Furthermore, since we already constructed the strong components, we can search for possible algebraic loops in the CBD. If this is the case we can already transform these algebraic loops into a proper structure as explained in Section 2.3. Before the actual compilation starts, the necessary data is gathered in dictionaries that will be used to properly construct either the LaTeX file or the DEVS file.

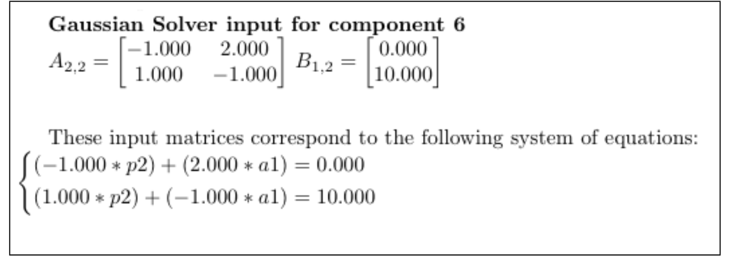
The initial design, provided in the *Reading Report*, showed a general structure of a transformation scheme from CBD models to LaTeX text files or to DEVS models. This structure however was based on the implementation of the compiler using AToMPM. As already noted in Section 1, the compilation of CBD models is actually implemented using Python, starting from the Python implementation of CBD's. Therefore, these structures are no longer valid, as the "intermediate language" consists of a dependency graph along with a series of dictionaries, rather than a model of a dependency graph. However, the general idea for the design remains the same. From a CBD model a dependency graph is constructed. Using this dependency graph the LaTeX file or the DEVS python file is generated according to the respective design described below.

3.2. Design of the LaTeX Component

The generated LaTeX file will consist of three distinct parts. First, the Causal Block Diagram is described. This descriptions includes a table that shows for each block in the Causal Block Diagram its name and type, a system of equations that describes the Causal Block Diagram and lastly a graphical representation. Next, a description of the Dependency Graph will be provided which contains a table that shows the different components along with the blocks that belong to this component, a table that shows the dependencies between the blocks in the Causal Block Diagram and a graphical representation of the Dependency Graph. Lastly, a solution of the Causal Block Diagram will be displayed. For this last section, three distinct options are available:

1. The Causal Block diagram did not contain any loops: In this case, a solution will be shown that displays for each block its signal value
2. The Causal Block diagram did contain loops, but all loops are linear: In this case the matrices that are used as input for the Gaussian Solver are shown, as well as their corresponding system of equations. A table containing for each block its signal values is displayed as well.
3. The Causal Block diagram did contain loops, at least one loop was non linear: A warning message is displayed that the Causal Block diagram contained a non linear loop and a solution could not be found.

Most information in this document is straightforward. Figure 2 and 3 contain an example of a Causal Block diagram with a linear loop and shows the resulting matrices along with their system of equations.


Gaussian Solver input for component 6

$$A_{2,2} = \begin{bmatrix} -1.000 & 2.000 \\ 1.000 & -1.000 \end{bmatrix} \; B_{1,2} = \begin{bmatrix} 0.000 \\ 10.000 \end{bmatrix}$$

These input matrices correspond to the following system of equations:

$$\begin{cases} (-1.000 * p2) + (2.000 * a1) = 0.000 \\ (1.000 * p2) + (-1.000 * a1) = 10.000 \end{cases}$$

Figure 2: Gaussian Solver Input Matrices example

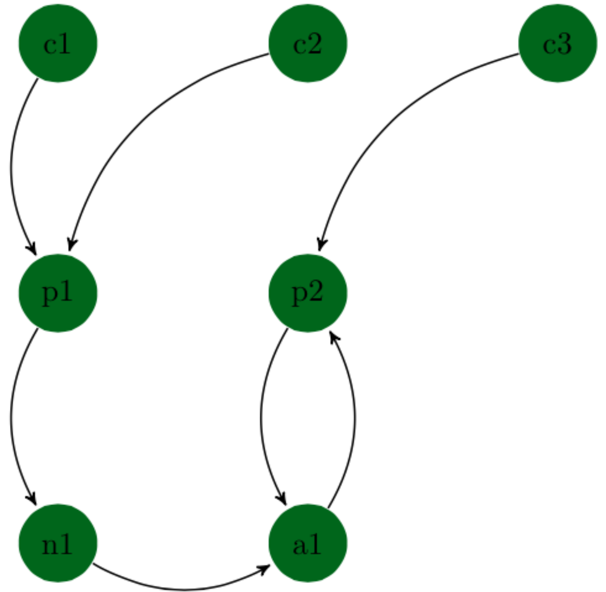


Figure 3: Causal Block Diagram graphical representation

3.3. Design of the DEVS Component

The generated Python file containing the AtomicDEVS model of the Causal Block diagram consists of the following parts:

- **CBDState**: The state definition of the AtomicDEVS which contains a variable for each block. These variables hold the signal value of the block. This class will be used to store the values of the blocks as the simulation runs. Values are updated at appropriate times and a general description of the state of the Causal Block Diagram can be accessed via the "to string" method.

- **CBD**: The AtomicDEVS that conforms to the Causal Block Diagram model. This AtomicDEVS will contain a state for each component (there are some exceptions, see Section 4). Inside the internal transition method, the appropriate block values will be computed and updated before proceeding to a next state.

- **Root**: The Root class is a CoupledDEVS that holds the CBD AtomicDEVS model as only submodel. It is used as input for the simulator and will have no other purpose.

Besides the standard methods that are available in state definition classes and AtomicDEVS classes the following methods will be added to allow the computation of linear loops:

- **getValue**: This method, present in the class CBD-State, returns for a certain block name (as string) its corresponding value. It will be used to construct the input matrices M1 and M2 described in Section 2.
- **constructLinearOutput**: This method, present in the class CBD, computes the matrices M1 and M2 used for solving a linear algebraic loop.
- **gaussJLinearSolver**: This method, present in the class CBD, solves the linear algebraic loop based on the input matrices M1 and M2.

4. Implementation

In this section I will discuss the implementation of the CBD compiler. I will discuss the different constructs that were used to generate the ".tex" and ".py" files as well as some implementation decisions I made.

The implementation itself is written in Python and uses the CBD model provided in the course Modelling of Software-Intensive Systems as starting point for compilation. In both cases, the dependency graph is constructed using the available methods in the CBD implementation. The file generation is performed using the built-in File functionality of Python.
I will now divide my discussion, as both compilers – CBD to LaTeX and CBD to DEVS – are constructed very differently. In both cases I will try to follow the structure of the corresponding file to provide decent structure.

4.1. CBD2LatexCompiler

A first step in the compilation process is the verify whether certain properties of the Causal Block Diagram are true. First, the CBD is checked for hierarchy. If the CBD is hierarchical it is flattened, such that all its contents are easily accesible, rather than having to use recursive methods to retrieve all blocks. Next we check whether the CBD contains a Derivator- or Integrator-Block. This check is performed only to determine how many iterations should be displayed in the solution. If it contains no such blocks, only a single iteration suffices, as the signal values will not change in the next iteration. If these blocks are present, multiple iterations are displayed as values can change.

The graphical representation in LaTeX is realised using the *tikz* package. The next step in the compilation process will thus be to setup a dictionary called **tikzLevels**. This dictionary contains blocks that should be displayed, organized in a level-like fashion. A level represents a horizontal line of blocks. I use this setup to limit the amount blocks that are horizontally aligned. Otherwise, a part of the graph could fall ofscreen. The levels are determined based on dependencies. Blocks in level 1 will have no influencers, blocks on level 2 are influenced by blocks on level 1, etcetera. This is not necessarily an ideal layout mechanism, but it limits the amount of edges crossing each other.

The next step in the compilation process is to construct the textual description of the CBD. Here a table is constructed using the blockname and blocktype of each block. Furthermore, a system of equations is constructed that describes the CBD. For each block, its equation is constructed based on its influencers. Of course, in the case of Constant- or TimeBlocks, this equation corresponds to "being equal to a value".

The graphical representation is built based on the dictionary **tikzLevels** and the dependencies between the blocks. This information is entered in a tikz specific format, which produces a graph based image.

Next, the information concerning the dependency graph is displayed. This is performed in the same way as for the CBD. Here the information consists of the components and their blocks on one hand, and blocks and their dependencies on the other hand. The graphical representation looks similar to the graphical representation of the CBD, but the dependencies are mostly in reversed order.

The last and most interesting section is constructed based on the properties of the CBD. If the CBD contains no loops, a single table is constructed displaying the signal values of each block (the number of iterations depends on the type of blocks in the CBD, see above). If the CBD did contain loops and at least one loop was nonLinear, then no specific information is constructed. In this case a single message is produced.
If the CBD contained loops which are all linear, the input matrices M1 and M2 are gathered from the CBD per component and displayed in the document. For each set of matrices, the equivalent system of equations is constructed.

4.2. CBD2DevsCompiler

The CBD2DevsCompiler contains some additional methods that aid in the compilation process:

- **allInfluencersKnown**: Used to determine which block values can be determined from the initial state of the CBD without actually running the CBD model

in a simulator. This is performed using value propagation as far as possible.

- **getActionStatement**: Returns the update statement to compute a block's value inside the action methods inside intTransition.

- **getBlockValue**: Used to initially set the block values where possible in the CBDState. This method will only be called if the value can actually be computed at the initial state of the CBD, i.e. when the method allInfluencersKnown returns True.

Using these methods the DEVS file is generated as follows.
First the same checks are made as in the CBD2LatexCompiler (Hierarchical, ContainsLoop). Next, after the preface with the standard import statements, 4 dictionaries are constructed that will be used in the methods related to computing a linear algebraic loop.

1. **dependsDict**: Contains for each block all the blocks it depends on. The list of blocks contains for each block its name and can be used to ask the state for the value of this block.
2. **influencesDict**: Contains for each block all the blocks it influences.
3. **componentDict**: Contains for each strong component in the dependency graph the blocks that belong to this component. Entries of this dictionary is used as input for the *constructLinearOutput* method
4. **typeDict**: Contains for each block its block type. This dictionary is used in the *constructLinearOutput* method to verify which block type a certain block has.

Next we construct the CBDState class. As stated before, for each block whose value can be determined at the initial state of the CBD, we enter this value in the CBDState as it will never change. An exception here is that we do not set the value for a TimeBlock and DelayBlock as fixed, as they will change over time. All other blocks are initialized using dictionary entries that are provided by the action methods in the intTransition method. The method **getValue** in CBDState is added to allow the method constructLinearOutput to retrieve a value from CBDState based on the block name.

The next step in the compilation process is to setup the states of the CBD class. The state "start" is always present in each CBD. Furthermore, for each strong component where the values of the blocks could not be determined at setup, we add a new state. We check whether these values are known by comparing them to a list "knownblockvalues" which contains all blocks that are known at setup. Also we do not add a separate state to the CBD AtomicDEVS for TimeBlocks, as their value is only updated when we return to the start state (i.e. when all other blocks are computed and a time step is completed).

Next we construct the CBD AtomicDEVS class. We start by providing the initializer which initializes the state as the start state with all unknown values marked as None. Next we add the methods timeAdvance and outputFunc which respectively return the rate of the CBD and an empty dictionary since the CBD does not produce any direct output. Next we generate the intTransition method which has a condition and action subfunc for each state transition. A state transition always happens between two consecutive components for which states were created. From the start state we go to the first component and from the last component we return back to the start state. The action for each transition will be to compute the block values inside the component represented by the state we are transitioning from. The new state, were we are transitioning to, will thus contain the updated values for these blocks. Inside the condition we will always verify that, for all influencers of each block in the component the block value is not None (as otherwise we are not able to compute its value). If these values are None, we return None as new block value.
In the case of a linear algebraic loop, the block values are computed using the method **gaussjLinearSolver**. Next we provide the method extTransition, which returns an empty dictionary as we do not use external transition in this model. Lastly we provide the methods **updateRate**, **constructLinearOutput** and **gaussjLinearSolver** which are already discussed earlier. Lastly we construct the fixed Root class.

An important implementation detail that I still need to discuss is how the values for DelayBlocks are handled. During initialization, the value for the DelayBlock is initialized as its initial condition. Inside the intTransition method, the value for the DelayBlock is not updated in the state of its corresponding component, but it is updated when its influencer is updated. When we compute the new value for the influencer, the value for this influencer in CBDState is still its old value. Therefor we update the value of the DelayBlock with this old value and then return both the new value for the influencer as the new value for the DelayBlock.

5. Results

Along with this project, a series of test files are provided which can be used to produce AtomicDEVS and ".tex" files from a number of CBD models. When simulating these AtomicDEVS models, we clearly see that the results are similar to the result of simulating the CBD model. Furthermore, when rendering a pdf from the ".tex" files we clearly see that the information that is displayed is correct. Furthermore we see that the compilation of the CBD models to both LaTeX and DEVS runs reasonably fast. The test suite, which generates a LaTeX file and a

DEVS model for 6 different CBD models, completes on average in 0.165s[7]

6. Conclusion

Even though the original implementation was planned using the AToMPM modelling tool, the resulting implementation appears to be sufficiently stable and easy to use. Both for generating LaTeX files and DEVS models, the compilation process can generate properly working files without any user interference. Both in the case of CBD models without loop and CBD models with linear algebraic loops, the DEVS model computes the correct output values. Furthermore we find that the LaTeX files contain the correct information in all cases.

Future work for both compilers could include an automated monitoring element for dynamically changing the rate. Furthermore the inclusion of logical operations in both compilers could be applied. Further optimizations could also consist of optimizing the data storage of block values in the DEVS model.

References

Bolduc, J.S., Vangheluwe, H., 2002. Expressing ode models as devs: Quantization approaches. quantum 2, 1.

Bolduc, J.S., Vangheluwe, H., 2003. Mapping odes to devs: Adaptive quantization, in: Summer Computer Simulation Conference, Society for Computer Simulation International; 1998. pp. 401–407.

Christis, N., 2012-2013. Research internship 2: Hybrid systems.

EncyclopediaOfMath, 2011. Linear equation. URL: `www.encyclopediaofmath.org`.

de Lara Jaramillo, J., Vangheluwe, H., Moreno, M.A., 2003. Using meta-modelling and graph grammars to create modelling environments. Electronic Notes in Theoretical Computer Science 72, 36 – 50.

Posse, E., Lara, J.D., Vangheluwe, H., 2002. Processing causal block diagrams with graph-grammars in atom.

Van Mierlo, S., Van Tendeloo, Y., Mustafiz, S., Barroca, B., 2014. Debugging parallel devs.

Vangheluwe, H., 2012. Model transformation.

Zeigler, B.P., Praehofer, H., Kim, T.G., 2000. Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems. Academic press.

[7] tested on a Macbook Pro with a 2,9 GHz Intel Core i7 processor.

Appendix A. DepGraph algorithms

Listing 1: Topological Sort

```
# topSort() and dfsLabelling() both refer
# to global counter dfsCounter which will be
# incremented during the topological sort.
# It will be used to assign an orderNumber to
# each node in the graph.
dfsCounter = 1

# topSort() performs a topological sort on
# a directed, possibly cyclic graph.
def topSort(graph):

        # Mark all nodes in the graph as un-visited for node in graph:
        node.visited = FALSE
        # Some topSort algorithms start from a "root" node
        # (defined as a node with in-degree = 0).
        # As we need to use topSort() on cyclic graphs (in our strongComp
        # algorithm), there may not exist such a "root" node.
        # We will keep starting a dfsLabelling() from any node in
        # the graph until all nodes have been visited.
        for node in graph:
                if not node.visited:
                        dfsLabelling(node)

        # dfsLabelling() does a depth-first traversal of a possibly
        # cyclic directed graph. By marking nodes visited upon first
        # encounter, we avoid infinite looping.
        def dfsLabelling(node, graph):
                # if the node has already been visited, the recursion stops here
                if not node.visited:
                        # avoid infinite loops
                        node.visited = TRUE
                # visit all neigbours first (depth first)
                for neigbour in node.out_neigbours:
                        dfsLabelling(neighbour, graph)
                # label the node with the counter and
                # subsequently increment it
                node.orderNumber = dfsCounter
                dfsCounter += 1
```

Listing 2: Strong Components

```
# Produce a list of strong components.
# Strong components are given as lists of nodes.
# If a node is not in a cycle, it will be in a strong
# component with only itself as a member.
        def strongComp(graph):
                # Do a topological ordering of nodes in the graph
                topSort(graph)

        # note how the ordering information is not lost
        # in subsequent processing and will be used during
        # Time Slicing simulation.
        # Produce a new graph with all edges
```

```
reversed. rev_graph = reverse_edges(graph)

# Start with an empty list of strong components
strong_components = []

# Mark all nodes as not visited
# setting the stage for some form of dfs of rev_graph
for node in rev_graph:
        node.visited = FALSE

# As strong components are discovered and added to the
# strong_components list, they will be removed from rev_graph.
# The algorithm terminates when rev_graph is reduced to empty.
while rev_graph != empty:
        # Start from the highest numbered node in rev_graph
        # (the numbering is due to the "forward" topological sort
        # on graph
        start_node = highest_orderNumber(rev_graph)

        # Do a depth first search on rev_graph starting from
        # start_node, collecting all nodes visited.
        # This collection (a list) will be a strong component.
        # The dfsCollect() is very similar to strongComp().
        # It also marks nodes as visited to avoid infinite loops.
        # Unlike strongComp(), it only collects nodes and does not number
        # them.
        component = dfsCollect(start_node, rev_graph)

        # Add the found strong component to the list of strong components.
        strong_components.append(component)

        # Remove the identified strong component (which may, in the limit,
        # consist of a single node).
        rev_graph.remove(component)
```