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 verify whether the model behaves correctly.

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 will be developed using the AtomPM modelling tool. Both for *LaTeX* and *DEVS* a certain transformation scheme will be provided. This transformation scheme will be used to construct a DEVS model in AtomPM on one hand, and an exportation mechanism for generating the LaTeX document on the other hand.

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[1], 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.

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[2]. Using these elements the main concept of this aspect will be to create an Atomic DEVS element for each strong component in the CBD model. The idea behind this concept is that in

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

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

order to work with algebraic loops, these loops should be treated as a single element. More on this in Section 2 where I will discuss the theoretical background for this compiler.

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. Lastly, in Section 5 I conclude on the project and will present some possible Future Work.

Note that I do not yet include Section 4 in my structure overview. This report is only a first version of my project report and it consists of performing a problem analysis and providing a design for the compiler. In the next version of this report, I will include the necessary implementation details. This next version will be ready once the implementation is completed.

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 be a model of a dependency graph. This means that we need to provide the proper algorithms to transform the source CBD model into such a dependency graph. Why such a model 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*[3]. The main purpose of using these algorithms is that we can create a dependency graph for the CBD model where:

- Blocks are represented by nodes where each node has a marking. This way we can look for a "root" node from which we can start our simulation in DEVS for example.
- Strong components are already identified in order to find an early solution to algebraic loops.

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 model is built from the CBD model which serves as an intermediate language in the transformation scheme. The mapping a CBD to a dependency graph results in the following structural elements:

- *Vertices*: A node - which is a representation of a vertex in the graph - contains a link to the corresponding CBD block. Furthermore a node contains a marking for this block based on the topological sort algorithm. Lastly a node will contain a link to the strong component to which it belongs.
- *Edges*: The edges in this dependency graph will be directed and corresponds to the output-to-input links between different blocks in the CBD model.

2.2. CBD to DEVS mapping

As can be found in the article about debugging parallel DEVS by Van Mierlo et al. (2014), an Atomic DEVS model M can be written as $M =< X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$[4] where

1. X	Input Set
2. S	State Set
3. Y	Output Set
4. $\delta_{int} : S \to S$	Internal Transition
5. $\delta_{ext} : QxX \to S$	External Transition
6. $Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)\}$	total state, e is elapsed time
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 Atomic DEVS. Each state of this Atomic DEVS will thus represent a single block in the strong component[5].

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

[4] The transition type δ_{conf} is not included here even though it was included in the article by Van Mierlo et al. (2014). These transitions, also know as *confluent* transitions are used to decide which transition (internal or external) is fired in case of a conflict. It has a default setting in which the internal transition will be fired first. I will not change this behaviour in my model.

[5] As can be seen later, when an algebraic loop occurs in the CBD model, the blocks that belong to this loop will be treated as a single block. So in case of an algebraic loop, a state in an Atomic DEVS will actually represent multiple blocks of the CBD model

For each Atomic DEVS that is generated by the transformation we see that $X = Y = \{Signal\}$ where Signal is an event in our generated DEVS model that corresponds the signals between transitions. Such a signal carries the value that is outputted by a certain block in case of the original CBD model and is used as input value for another block.
The state space S of the Atomic Devs will consist of all the states that correspond to the blocks in the CBD diagram that belong to the strong component which is transformed in this specific Atomic DEVS.
The internal transitions of the Atomic DEVS will corresponds to the links between the blocks of original strong component to which the Atomic DEVS corresponds. They will thus "carry" the signal from one block in the strong component to another.
The external transitions also corresponds to links between blocks in the original CBD. However they do not corresponds to links between blocks in the same strong component, but they will corresponds to links between blocks that belong to different strong components.
The total state of the Atomic DEVS will consist of the output signal of each block represented by a state in the Atomic DEVS 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 is dependent on the links between the blocks in the CBD. Only those states that represent a block whose outgoing links arrive in another block that does not belong to the same strong component will generate events. These events are all of the type *Signal* (see earlier).

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 $a_1x_1 + a_2x_2 + \cdots + a_{N-1}x_{N-1} = b$ 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 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.

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 and by treating this system as a single block with multiple inputs and outputs. Depending on which input is present, a certain output can be generated. Or visa versa, if one wants to

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

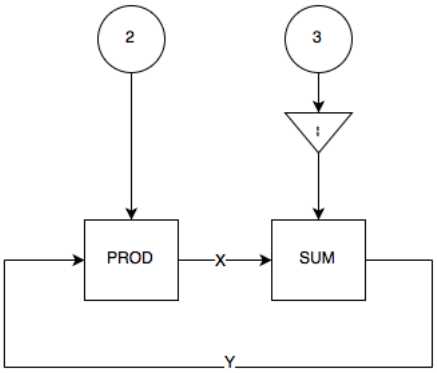

Figure 1: Example of a basic algebraic loop in CBD.

know a certain output value of the system we will need to look at the different input values from which this output is computed. To continue our previous example, if we transform this algebraic loop into a single block, we would have x, y as input and as output set. In the example above, as there are no inputs except the ConstantBlocks, it can be easily calculated that by solving the system of equations the output signal for x will always be 6 and the output signal for y will always be 3.

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. If time allows I will extend this section with details on how exactly this dynamic block rate change could be designed.

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 model. 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.

3.2. Design of the LaTeX Component

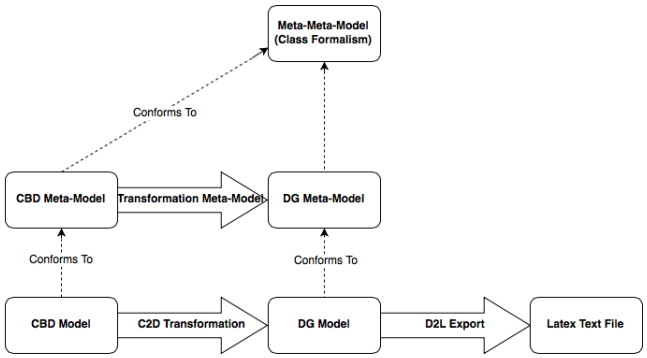

Figure 2: Structure of the CBD to LaTeX transformation scheme

Figure 2 shows the general structure of the transformation scheme from CBD models to LaTeX file. A CBD model is first transformed into a Dependency Graph Model (DP Model) as was mentioned earlier. The DP Model is then in turn exported to a file. Note that I did not yet write LaTeX file. The exportation functionality will not immediately generate a LaTeX file. It will however generate a MetaDepth file containing the different constructs that will be present in the resulting LaTeX file. Using this metaDepth model file we will generate the actual LaTeX file using the *Epsilon Generation Language*[7].

The result of this operation will be something similar to the following:

Listing 1: LaTeX output for the Algebraic Loop example

```
\documentclass{article}
\begin{document}

\section{Blocks}
% Information about the blocks and
% their links
\begin{tabular}{| c | c | c | c |}
\hline
```

[7] http://eclipse.org/epsilon/doc/egl/

```
Name & Type & Inputs & Outputs\\
\hline
\hline
b1 & ProductBlock & (b2, b3) & (b2)\\
\hline
b2 & AddBlock & (b1, b4) & (b1)\\
\hline
b3 & ConstantBlock & () & (b1)\\
\hline
b3 & ConstantBlock & () & (b2)\\
\hline
\end{tabular}

\section{Dependency Graph}
% If possible this will be done visually
% with the package Tikz,
% otherwise textual
\subsection{Strong Components}

\subsection{Algebraic Loops}
%Information about algebraic loops
% is instered here
\end{document}
```

The resulting LaTeX file in Listing 1 is of course an example and is not written in stone. It is to early to state which will be displayed exactly and in which order, but this example should already give a general idea.

3.3. Design of the DEVS Component

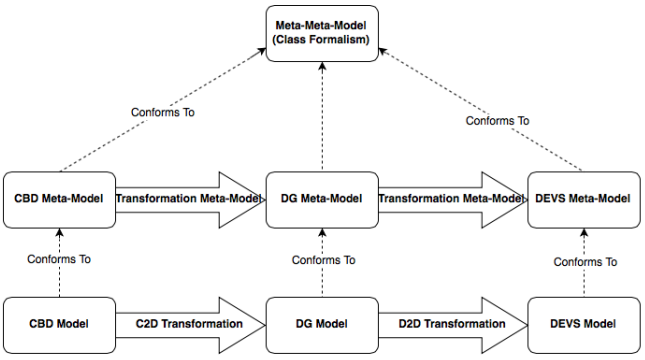

Figure 3: Structure of the CBD to DEVS transformation scheme

Figure 3 shows the general structure of the transformation scheme from CBD models to DEVS models. As can be seen in this Figure, the transformation from CBD to DEVS operates for the most part (the first step) identically to the transformation from CBD to LaTeX introduced in Section 3.2. The most important difference is of course that we will not export the Dependency Graph to some metaDepth model, but perform yet another transformation in AtomPM from the DP model to a DEVS model. This last transformation will involve mapping the different strong components to Atomic DEVS and providing other elements such as a Coupled DEVS which contains instances of the Atomic DEVS. Another element that should be created is the event Signal used for passing signals (read values on links) from one Atomic DEVS to the other.

The last part that should be provided is a time advance function that specifies after how much time such a signal should be generated. This will of course depend both on input signals as well as the rate of the corresponding block(s). The details on how this will all work will become clear once I will be able to discuss the implementation.

Now I will provide a description of the DEVS model that will result from the Algebraic Loop example in Section 2.3.
The resulting DEVS model will only contain a single Atomic DEVS as their is a single strong component in the dependency graph of the CBD model. This Atomic DEVS will have two output ports, "X" and "Y" (one for each output signal). As we can already - and easily - solve the algebraic loop in this strong component, the Atomic DEVS will only contain a single state that represents the system of linear equations that corresponds to the algebraic loop. Since we have shown in Section 2.3 that X will always equal 6 and Y will always equal 3, we see that this single state will generate Signal events for both output ports (one containing the value 6 for the output port "X", and one containing 3 for the output port "Y"). There are no external transitions needed (as there is only a single Atomic DEVS). There is however a single internal transition which will go from this one state to itself. This way, we keep generating events (every time the time advance constraint is met).

This easy example shows the basic scheme for the CBD to DEVS transformation. Of course the eventual compiler will generate much more complex DEVS models. But this example suffices for now.

4. Implementation

The contens of this section will be provided once the implementation is finished.

5. Conclusion

No real conclusion can yet be formed as we have not yet implemented the compiler itself and thus have not run a series of experiments. We can however already state that by first transforming the CBD model to an intermediate Dependency Graph model we will be able to extract additional information from the CBD such as strong components and algebraic loops. By already finding a solution for the algebraic loops, the resulting LaTeX file will contain

additional, interesting information. Moreover, the resulting DEVS model will be significantly easier, as the number of states and transitions will be reduced.

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.

Appendix A. DepGraph algorithms

Listing 2: 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 3: 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)
```