

From model to simulation, the a-causal way

Yves Maris

Abstract

Modelica has become an ad-hoc standard for a-causal modeling. The aim of this paper is to increase understanding of the language by giving a complete overview ranging from language features to model compilation. To illustrate this compilation process to simulate a model, a simple Modelica like compiler, for a-causal to causal block diagrams, is constructed and discussed.

Keywords: Modelica, mathematical modeling, Modelica compilation process, a-causal block diagrams, ABD, ABD compiler

1. Introduction

Block oriented languages have been around for quite some time. A well known example of this is the Simulink language introduced by Mathworks in 1993 (Simulink and Natick, 1993). In block oriented languages a hierarchical structure of individual blocks with defined input and output is build. The relation between different blocks is solely defined by the output of one block being fed into the input of an other block. This is why this kind of modeling is called causal. Yet there is another way describe individual parts of a model and that is trough equations. This means these components will not describe an output with a specific output anymore. This approach is called a-causal modeling. There has been a number of a-causal modeling languages but this paper will focus on Modelica. In the first section an overview will be given of these other languages and how they relate to Modelica. The second section will go more in depth on the features of the language. More specifically an overview of the object oriented constructs and the manner in which mathematical modeling is made possible will be discussed. After that an overview of the Modelica compilation process is given in which each

Email address: `yves.maris@student.uantwerpen.be` (Yves Maris)

step from model to simulation is discussed. The theory explained in those sections will be put in to practice by building a Modelica like compiler for a-causal block diagrams that will generate the simulation code in a Modelica like manner.

2. Related work

Modelica isn't by far the first a-causal modeling language. Languages like Dymola (Otter et al., 1996), NMF (Vuolle and Bring, 1997) and Omola (Andersson, 1990) can be seen of the predecessors of Modelica. Some languages like NMF continued to be used in specific domains like building similitions Vuolle and Bring (1997) while others disappeared completely. Dymola is now a commercial modeling and simulation environment based on the Modelica language, it no longer implements the Dymola language. This tool even makes it possible to export modelica models to the Simscape models and inversely, import simscape models as Modelica models. One of the main competitors of Modelica now is Simscape by The Mathworks. It was developed as an add-on for Simulink which uses the Matlab language. Simscape differs from Modelica in the compilation process which leads to some variability in efficiency. Modelica compilers might be more efficient in most cases when model fragmentation is present the Simscape compiler has an advantage. A more in depth explanation of this topic can be found in Section 4. An other example of a modeling language similar to the Modelica language is EcosimPro, developed by Empresarios Agrupados A.I.E for the European Space Agency. This language was originally developed for simulating spacecraft systems though it can be used for more general purpose applications Laurini et al. (1999). There are a lot of similarities between these tools because most of them are in some way based on the phd thesis of Hilding Elmqvist (Elmqvist (1978)).

3. An overview of Modelica

The aim of this section is to give a basic understanding of the capabilities and constructs of Modelica. This means no detailed description will be given on how to use the language. An excellent tutorial complete with examples is however given by Elmqvist et al. (1999) on which this section will be based.

Modelica is an object oriented modeling language designed for a-causal modeling although it also supports causal models. This makes it possible to

model complex systems in an intuitive way. When it is too expensive, complex or dangerous to construct a prototype of a real world system it might be appropriate to make a virtual model instead. A system in Modelica is an object or a collection of objects of which we want to study the behaviour by running experiments. These systems can be either natural or artificial of nature. The basics of Modelica is that you build components that interact with each other. Each of these components can be build up from other components. This concept is called hierarchical modeling. Physical connections between components allow interaction between them. A common misconception about Modelica is that it is a domain specific modeling language. This is not true since the language has no definition on how to impose constraints to models.

3.1. Object oriented

Modelica differs from traditional object-oriented programming languages in a sense that it uses the object orientation aspect mainly as a structuring concept (Fritzson (2010)). This while the traditional languages also use it for dynamic message passing. This section will explain which concepts of object oriented languages are used within the language.

3.1.1. Classes

Everything in Modelica is modeled as a class ranging from physical connectors to functional components. This has some great advantages for the modeler since he only has to learn the features of this class to be able to use the language. In some cases it might not be desired to make use of the full functionality of this class concept. For this Modelica offers a set of seven restricted class types. Each of these restricted classes is defined by it's own keyword that can will be used instead of the `class` keyword. A small overview of these restricted classes and the imposed restrictions is given below.

- **Type:** A class definition in Modelica creates a type name for that class. This restricted class can only define such a type name using the build-in classes like for example `Real`.
- **Connector:** This class exists for defining physical connections.
- **Model:** Can not be used as a connector.

- **Record:** This class can only contain data and no equation. It can be thought of as a parameter set.
- **Block:** A class with input-output causality (For creating causal models).
- **Package:** Can only contain declarations of constants and classes. This is used to group these related classes together to create structure and avoid so called “name clashes”.
- **Function:** A class with input arguments that is dynamically instantiated when called.

3.1.2. *inheritance*

One of the great strengths of object oriented languages is inheritance. In other words the content and behaviour of the superclass will be copied to subclass. This concept is fully supported within the Modelica language. This is done by the keyword `extends` followed by an optional access specifier followed by the superclass. Multiple inheritance is also supported, a common problem with this is so called diamond inheritance where two classes used for multiple inheritance have the same base class as seen in Figure 1. In c++ for example this problem is solved by virtual base classes. In Modelica it is even simpler when the scenario in Figure 1 occurs simply one instance of the base class `a` is kept. No explicit action is needed. It is also possible to define interfaces for a class. This can be done by using the keyword `partial` in front of the interface specifying that the class definition is incomplete.

3.1.3. *Information hiding*

In the default behaviour of Modeling all variables are accessible from the outside via the dot operator. Information hiding is crucial to create higher maintainability. To enforce this Modelica has one access specifier namely `protected`. `protected` members can still be accessed from subclasses of the respective class in which the variable is defined. Yet no other access specifiers are defined to maintain simplicity.

3.2. *Mathematical modeling*

3.2.1. *Variable types*

The functioning of Modelica is based around two different types of variables. Each of this variables has a different behaviour when a connection is

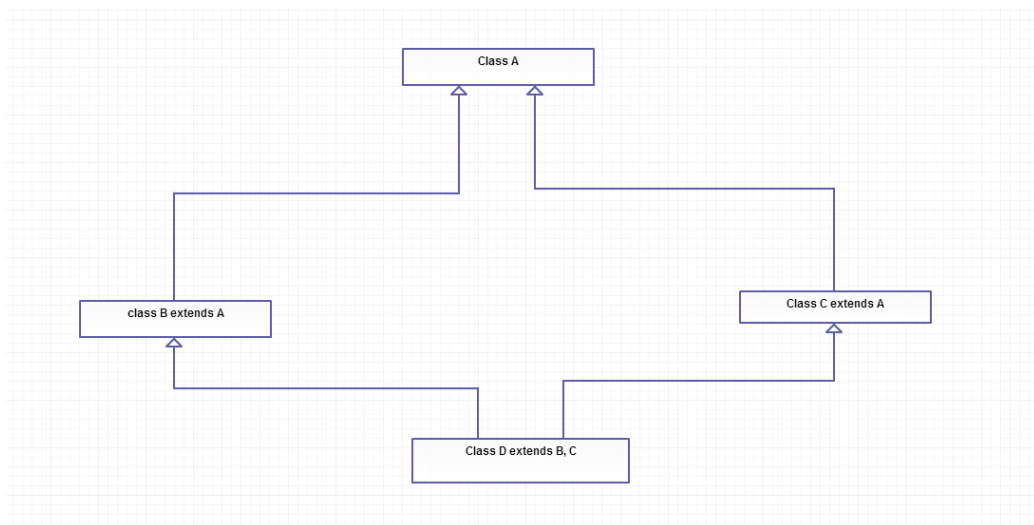


Figure 1: Caption

made. The first type is trough variables, they were added to provide a way to follow Kirchoff’s current law, as stated in Definition 1. This law states that the sum of all currents in one component should be zero and similar laws apply to fluids and other systems. This must be specified by adding the keyword flow in front of the variable declaration. The other kind of variables are across variables. This is the default variable types are connected across variables are set to equal.

Definition 1 (Kirchoff’s current law). $\sum_{i=k}^n I_k = 0$ With n the number of incoming and outgoing connection.

3.3. Equations

Equations are used to express relations between variables. Do not confuse equations with assignment statements. There is no notion of assignment within equations also there is no particular order in which they are evaluated. Both the left hand and the right hand side of an equation can contain expressions. Modelica does define specific sections for assignment statements combined with control structures. These sections are called algorithm clauses.

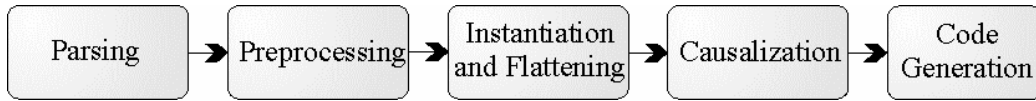


Figure 2: The compilation steps taken by a Modelica compiler

4. The Modelica compilation process

In this section a description will be given of the compilation process by a Modelica compiler (Fritzson et al., 2002). The general steps of used by for example Open Modelica and Dymola will be discussed, this process is graphically represented in Figure 2 Zimmer (2009). The compilation starts by flattening the model, in this step all object oriented structures are removed and a collection of differential and algebraic equations is left. A causalisation step is then applied to the resulting flat model. In this step is implemented a-causal equations are converted to causal statements. This is also where the Modelica language takes an other approach than Simscape. There is no causalisation step in Simscape equations are resolved by a numerical solver. This means Modelica is in a disadvantage when it comes down to dynamic models. If a model changes the causalisation step has to be repeated for the new model(s). The compilation process will be fully illustrated in Section 5.

5. Implementation

To provide greater insight of how a a-causal model compiler functions, a simple one will be implemented. The input of this compiler will be a a-causal block diagram (ABD). Before any other steps are taken the model is flattened as described in Subsection ???. A a-causal block diagram consists of multiple blocks connected by links. Yet because the model is a-causal these links are not directional. The first task to make the model causal is to assign IO causality to the ports of the model in order to make the links directional. Each of the a-causal blocks corresponds to a certain mathematical operator, as seen in Section 5.2. Assigning IO to the blocks does not necessarily match the a-causal block with it's causal equivalent. This is why some transformation is necessary to achieve an equivalent meaning for the resulting causal block diagram. This last step in the compilation process is discussed in Section 5.4.

A different compiler is also created in order to be able to represent the different constraint equations that describe the ABD model in the form of a

latex file. This is done by assigning a unique name to each link in the ABD. These names are represented by an upper case alphabetical letter followed by a optional number. Some ports are not included in a link because it they are used as an input or output for the entire ABD. In this case the corresponding variable is displayed as `block_name(port_name)`. An overview of the equations described by these variables is given in Section 5.2.

5.1. Flattening

In this implementation a choice has been made to first resolve all hierarchy before beginning the causalisation step. This approach is also taken by most Modelica compilers. It is important to mention that this does not mean that the resulting CBD will be flat. Hierarchy can be introduced again by transforming certain blocks in their causal version as seen in Section 5.4. The model is flattend by removing all InputBlocks and OutputBlocks from the child blocks and removing hem with wire blocks. In the flattening process these wire blocks also get their expected causality assigned to them to make sure no meaningless models are generated. The links of all parent and child blocks connected to these input and output blocks are also removed and replaced with the corresponding WireBlock links. The WireBlock is a simple block with two ports, P1 and P2, that describes the constraint equation $signal(P1) = Signal(P2)$. Every block that was part of the Childblock is also renamed to `name_child_CBD + _ + name_block`.

5.2. Assigning IO causality

Needless to say that the basics of a causal model is that the ports have IO causality. In an ABD there are some ports that have predetermined output causality:

- A constant block
- An input block

By starting from these output ports all connected ports can be labeled as input. If all ports of the block have been assigned input causality, the last port can just be labeled as an output port. This is because every block in this implementation has exactly one output port. An iteration in the IO causality assignment algorithm consists of checking this last statement and labeling a port as input when it is connected to a output port and visa versa. If the number of blocks with unknown causality stays the same for three iterations,

the algorithm fails and decides that no solution is possible. The algorithm is described in pseudo code as follows:

```

BlocksToAssign = every block with some port with
    unknown causality
while blocksToAssign:
for block in blocksToAssign:
    if only one port to assign:
        if output is known
            unknownPort = input
        else:
            unknownPort = output
    for port in block:
        if outputport connected to port:
            port = input
        elif inputport connected to port:
            port = output

```

For some ports a more “aggressive” approach is taken and the preferred IO is specified when the component is specified in the constructor, for example in the IC and `delta_T` ports of the Derivator and integratorBlock.

```
self._infoLinks["IC"] = causality.INPUT
```

5.3. Corresponding operators

Every a-causal block has a specific meaning that might differ from the causal meaning. These meanings can be found in Table 1. A brief discussion is given for the most important blocks.

AdderBlock. The meaning of an a-causal Adder block will differ quite a bit from that of the corresponding causal block. While the causal block has a number of predefined inputs and one output port. The incoming signals will be summed to the output signal that is then appended to the output ports signal. In the a-causal version on the other hand it is not possible to express the signal of one port as an operation applied to the signals of the other port since this implies that there is some causal relation. The relation between the ports has to be representable by some constraint equation. In the case of the adder block this can be done by summing all signals to zero. This also means that there will be no difference in the causal meaning when different input or outputs Causalities are assigned. We say that the order of the assigned

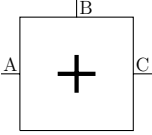
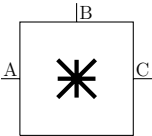
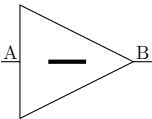
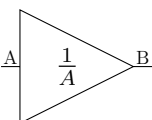
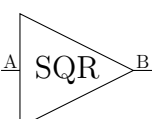
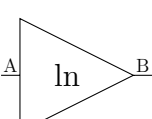
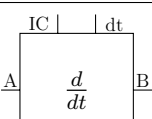
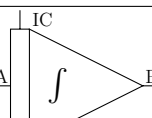
		$= A + B + C = 0$
		$= A * B * C = 1$
		$= A + B = 0$
		$= A + B = 0$
		$= A^2 = B$
		$= \ln A = B$
		$= \frac{d}{dt} A = B$
		$= \int dt A = B$

Table 1: The meanings of the most important a-causal block

ports does not matter for the assigned causality. This is why we will call this type of blocks unordered causal blocks.

NegatorBlock. This is a natural a-causal block, this means that there is a one to one mapping between the causal and the a-causal block. If we take a look at the a-causal blocks, the NegatorBlock can be represented as a simplification of the AdderBlock. If the number of ports of the Adder block is reduced to two, lets call them P1 and P2, than the equation that describes the relation between the these ports equals: $signal(P1) + signal(P2) = 0$. This can also be written as $signal(P1) = -signal(P2)$ witch is the causal meaning of the block.

ProductBlock. The product block can be approached in a very similar way as the adder block. The causal meaning of this block again is very similar to that of the adder block. In this block the product of all inputs is appended to the output signal. The connection of the a-causal block is represented by the constraint equation that takes the product of all variables and states tat this must equal zero.

InverterBlock. The InverterBlock is a second block that is natural a-causal. Here the block can be represented as a product block with two ports. If we call these ports P1 and P2 this gives the constraint equation $signal(P1) * signal(P2) = 1$. If this equation is converted to $signal(P1) = \frac{1}{signal(P2)}$ it becomes obvious that the a-causal meaning is the same as that of the corresponding causal block.

SQRBlock. The SQRBlock has two ports, P1 and P2, that represent the constraint equation $signal(P1)^2 = signal(P2)$. In contrast to the other blocks the choice of the input and output ports does change the causal meaning. The order of the ports does matter in this case, these type's of blocks are called ordered a-causal blocks.

LNBlock. Just like the SQRBlock, the LNBlock has two ports. The operator described by this block equals $\ln signal(P1) = signal(P2)$. This means two causal operations can be represented by this block, either the exponential operation or the natural logarithm. The choice of the input and output ports will completely determine which one of these two operations will be represented.

IntegratorBlock and DerivatorBlock. Both the a-causal IntegratorBlock and DerivatorBlock have 4 ports (IC, `delta_t`, P1 and P2). The IntegratorBlock describes the operation $\int signal(P1) = signal(P2)$ while the derivator block describes $\frac{d}{dt}signal(P1) = signal(P2)$. Although the a-causal operators are obviously different they can both represent the same causal operators. An IntegratorBlock with P2 as input and P1 as output is completely equivalent to a DerivatorBlock with P1 as input and P2 as output and visa versa.

5.4. Transforming to a CBD

As seen in Subsection 5.2 Some operators blocks need some transformation to make the a-causal block meaning match the one of the a-causal blocks. In this section these transformations will be discussed for the different types of blocks

Unordered a-causal blocks. For unordered causal blocks it does not matter which ports are identified as input or output, it will not change the corresponding causal block. This however does not mean that there is a one to one mapping between the causal and a-causal block. The AdderBlock and ProductBlock are the perfect example of this. If a correct IO causality is done the blocks will have one output port. If we try to convert this to it's corresponding CBD meaning we will actually get a small CBD. This CBD will contain a adder block and a NegatorBlock. The AdderBlock will get the two input signal and the result will be passed to the inverter block. This CBD will satisfy the constraint equation defined by the a-causal block. The following CBD is generated for the a-causal AdderBlock:

```
BlockName = CBD(block_name = BlockName, input_ports =
    ["IN1", "IN2"], output_ports = ["OUT1"])
BlockName.addBlock(AdderBlock(block_name="a1"))
BlockName.addBlock(NegatorBlock(block_name="n1"))
BlockName.addConnection("IN1", "a1")
BlockName.addConnection("IN2", "a1")
BlockName.addConnection("a1", "n1")
BlockName.addConnection("n1", "OUT1")
```

The ProductBlock also takes a similar approach. In this case the resulting CBD will consist of a causal Productblock connected to a inverterBlock.

Natural a-causal blocks. The NegatorBlock and the InverterBlock block are two examples of natural a-causal blocks. By definition this means that the a-causal meaning already matches the causal one. For every one of these blocks a equivalent causal block can be defined with the previously identified input and output ports without further transformation.

Unnatural a-causal blocks. We can speak of an unnatural a-causal block when an additional transformation step is needed to make sure the meaning of the resulting causal block is consistent with that of the original a-causal block. This might mean that one causal block is replaced with a network of causal blocks or that the a-causal block can correspond to multiple causal blocks. A distinction is made between two types:

Ordered a-causal blocks For ordered a-causal blocks the labeling of the ports as input or output will influence the corresponding a-causal block. This is the case for both the LNBlock and the SQRBlock. For example if port P1 is labeled as input and P2 as output, the resulting block will be product block that takes the same input signal for both ports. This represents a square operation which would result in the following output:

```
BlockName() = CBD(block_name = "BlockName", input_ports
    = ["IN1"], output_ports = ["OUT1"])
BlockName.addBlock(ProductBlock(block_name="p1"))
BlockName.addConnection("IN1", "p1")
BlockName.addConnection("IN1", "p1")
BlockName.addConnection("p1", "OUT1")
```

But when the causality is reversed and P2 becomes an input while P1 becomes an output, the corresponding causal block changes. It must now correspond to a block representing the square root operation. In the case of the CBD simulator in which this compiler was based this is a root block which takes the n-root of the input and n is an other input signal. To achieve the square root operation a constant block with value two was connected to the input representing n. This would result in the CBD below:

```
BlockName = CBD(block_name = "_BlockName_", input_ports
    = ["IN1"], output_ports = ["OUT1"])
BlockName.addBlock(RootBlock(block_name="r1"))
BlockName.addBlock(ConstantBlock(block_name="c1", value
    =2.0))
```

```

BlockName.addConnection("c2", "r1", input_port_name="IN2")
BlockName.addConnection("IN1", "r1")
BlockName.addConnection("r1", "OUT1")

```

A similar approach is again taken in for the LNBlock but the difference is that the causal GenericBlock is used to use the log and exp operations from the python library. These functions represent implement the ln and e^x operation. The DerivatorBlock and IntegratorBlock are both expanded in the cbd in this implementation. This means they are just compiled to a IntegratorBlock and a DerivatorBlock and not to a construction of other causal blocks. This allows these decisions to be made in the CBD compiler

6. Testing the compiler

The compiler described in Section 5 is developed in a test driven manner. This means a test suite is created to automatically verify the implementation. The python library “unittest” is used for the implementation. Since there are in fact two individual compilers, one generating latex code and the other a CBD model, two kinds of tests had to be created.

Latex tests. For each possible a-causal block a test is created. In each of these tests a minimal model is defined. Than the function `getLatex()` is called for that block to generate the latex output of interest. To determine whether this output is correct it is matched with a regular expression representing all acceptable outputs for the given model. To make this explanation more concrete a example is given for the InverterBlock

```

self.ABD = ABD(" ABD_for_block_under_test")
i1 = InverterBlock(block_name="i1")

self.ABD.addBlock(ConstantBlock(block_name="c1", value
    =5.0))

self.ABD.addBlock(i1)
    self.ABD.addBlock(InverterBlock(block_name="i2"))

self.ABD.addConnection("c1", "i1")

```

```

self.ABD.addConnection("i1", "i2")

p = re.compile('[A-Z]{1}\d*\*[A-Z]{1}\d* = 1')

self.assertTrue( p.match( i1.getLatex() ) )

```

The ABD described in this test is very simple It is just a constant block being connected to an InverterBlock that is connected to a second InverterBlock. In this test the output of the first one is tested. As seen in Subsection 5.2 the meaning of the inverter block is $A * B = 1$ if the connected link names are A and B. The regular expression $[A-Z]{1}\d**[A-Z]{1}\d* = 1$ simply says that the product of two link names is one as stated in by the meaning of this block. Similar approaches have been taken for the other blocks.

CBD tests. A completely different approach is taken to test the CBD compiler. This compiler does not only need to produce correct syntax, the resulting CBD also has to be meaningful. To make sure this is the case a (hierarchical) ABD model is created. This model is than compiled and the resulting CBD model is compiled using an appropriate compiler. The output of this execution is than captured and converted to a string. If this string matches the expected output, the compiler has created a valid model.

7. Conclusion and future work

To conclude this paper we give a brief summary of the most important concepts seen in this paper. Firstly in the study of the Modelica language we found found that the system behaviour was described by constraint equations. In the compilation process a Modelica model undergoes a flattening step before that flat model is causalised. This same process was implemented in the CBD to ABD compiler. After the flattening step each port is labeled as either input or output. Each of these blocks has a corresponding operator. Using the causal blocks with their corresponding operators the network is than transformed to a CBD with the same meaning as the ABD. To improve the compiler more extensive tests could be written to test for borderline cases. different versions of the IntegralBlock and DerivatorBlock blocks could also be made so their hierarchy is expanded in the ABD.

- Andersson, M. I., 1990. Omola: an object-oriented language for model representation.
- Elmqvist, H., 1978. A structured model language for large continuous systems. Lund Institute of Technology.
- Elmqvist, H., Boudaud, F., Broenink, J., Brück, D., Ernst, T., Fritzson, P., Jeandel, A., Juslin, K., Klose, M., Mattsson, S., et al., 1999. Modelicatm—a unified object-oriented language for physical systems modeling. Tutorial and Rationale, versión 1.
- Fritzson, P., 2010. Principles of object-oriented modeling and simulation with Modelica 2.1. John Wiley & Sons.
- Fritzson, P., Aronsson, P., Bunus, P., Engelson, V., Saldamli, L., Johansson, H., Karstöm, A., 2002. The open source modelica project. In: Proceedings of The 2th International Modelica Conference. pp. 18–19.
- Laurini, D., Thirkettle, A., Bockstahler, K., 1999. Supporting Life: Environmental Control and Life Support for the Multi-purpose Logistics Module (MPLM) of the International Space Station. ESA Publications Division.
- Otter, M., Elmqvist, H., Cellier, F. E., 1996. Modeling of multibody systems with the object-oriented modeling language dymola. *Nonlinear Dynamics* 9 (1-2), 91–112.
- Simulink, M., Natick, M., 1993. The mathworks. Inc., Natick, MA.
- Vuolle, M., Bring, A., 1997. An nmf based model library for building climate and energy simulation. In: Building Simulation'97-Fifth International IBPSA Conference.
- Zimmer, D., 2009. Module-preserving compilation of modelica models. In: Proc. of the 7th International Modelica Conference, Como, Italy.