Politecnico di Milano

Dipartimento Elettronica, Informazione e Bioingegneria

HEAP Lab Project Report

# VARIABLE FILTER for MODELICA ADVANCED RESEARCH COMPILER

Authors:

**Alessandro Lisi**
10621058

Supervisors:

Dr. Daniele Cattaneo

Dr. Federico Terraneo¶

## Abstract

MARCO is a Modelica compiler developed at Polimi by the PoliMi Modelica Working Group. Modelica is a modeling language for performing simulations in multiple engineering domains, which has seen increasing interest in the last decades. However, existing compilers for this language are not performant enough with respect to both temporal and spatial efficiency to deliver on the promise of easy enough simulation of modern complex and/or large scale systems. A **new Modelica compiler** named **MARCO** has been proposed as new approach to high-performance system simulation. The basic groundwork for the construction of this compiler is now complete and the team is now incrementally expanding its capabilities. The lowered model, represented as an equation system is translated to C code and compiled with a standard C compiler. Marco is based on the **LLVM** compiler infrastructure together with **Multi-Level Intermediate Representation** (MLIR).

### LLVM

LLVM is a collection of libraries built to support compiler development and related tasks. It has a modular design, this allows its functionality to be adapted and reused very easily.

### MLIR

The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together. MLIR facilitates the design and implementation of code generators, translators and optimizers at different levels of abstraction and also across application domains, hardware targets and execution environments.

One thing to keep is mind is that MLIR introduces a uniform concept called <u>operations</u> to enable describing many different levels of abstractions and computations.

*In MLIR “Everything is an Operation”*

The project is divided into tools and libs, which are two separate folders. Conceptually a lib does something and a tool lets you do the same things from command line. Code is written in C++, *MLIR tablegen* (a way of defining operations without using C++) has not been used by MARCO developers.

---

## 1. Introduction

The goal of this project is to extend Marco in order to support a **Variable Filter**. After using Marco to generate code and running a simulation, output is made indistinctly by all the variable declared in the model, array also are printed with all their dimensions and values. The role of the variable filter is to parse a command line argument, a string of semicolon separated tokens and build a representation of the provided input. This representation, stored in memory by a VariableFilter object will be later used to 'filter out' the values and the ranges of values that will be printed in future.

When using the VF it is possible to specify:

1. **variable identifiers**: print the variable that has the provided identifier

2. **array identifiers with ranges**: print an array, for each dimension of the array print only the values in the range specified for each dimension.

3. **regex**: print fully all the variables whose identifier matches one of the provided regular expressions

The project is thus made of a parsing phase, in which the provided string is parsed and the variable filter object is filled, and a code generation phase where the specified amount of 'prints' are generated by interrogating the Variable Filter (hereafter VF) instance.

---

## 2. Design and Implementation

By further analysing the requirements and the scope of the project, several steps of implementation can be outlined. They've been implemented this specific temporal order because by building a Gantt diagram of these task, the next is highly dependent on the completion of the previous.

In this section I'll discuss the design and programming of the variable filter and the way the marco MLIR codegen has been changed; its usage will be covered in dedicated sections.

First step is to fetch a string specified by command line arguments and parse it. Each token used to store information into the variable filter is separated by a ' ; ' so an independent parsing of each token is carried out.

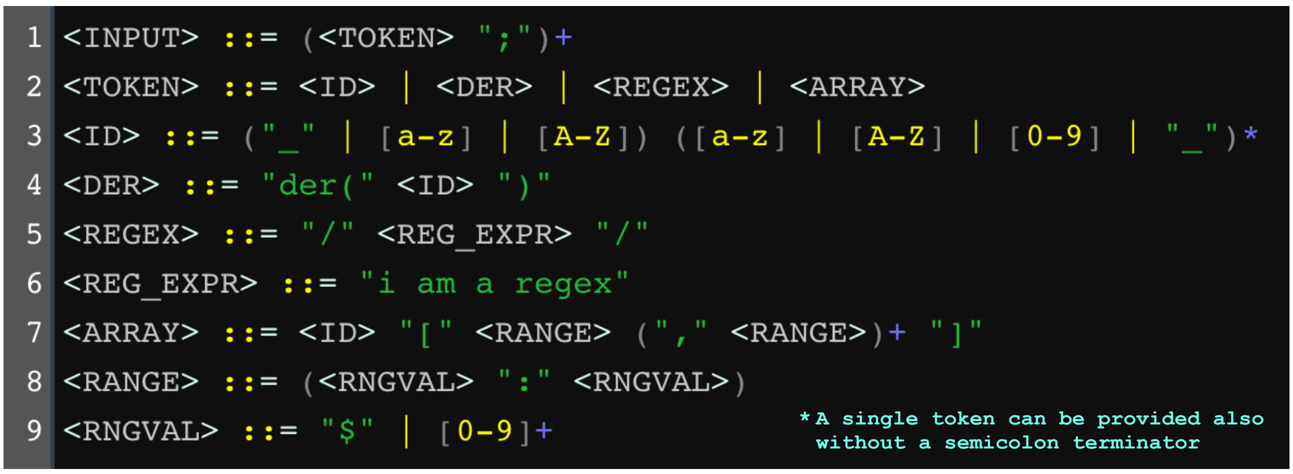
Because the following classes are two auxiliary tools that are/will be used in the several parts of MARCO, a reasonable choice was to insert this two files into lib/utils folder within `namespace marco`.

These classes have been defined in headers (.h) and implemented in source files (.cpp):

### 2.1 VariableFilterParser

This class is responsible, after providing a string token to be parsed and a VF instance, to parse the string and add the extracted information to the variable filter.

A BNF grammar on which the parsing is based is the following:



```
<INPUT> ::= (<TOKEN> ";")+
<TOKEN> ::= <ID> | <DER> | <REGEX> | <ARRAY>
<ID> ::= ("_" | [a-z] | [A-Z]) ([a-z] | [A-Z] | [0-9] | "_")*
<DER> ::= "der(" <ID> ")"
<REGEX> ::= "/" <REG_EXPR> "/"
<REG_EXPR> ::= "i am a regex"
<ARRAY> ::= <ID> "[" <RANGE> ("," <RANGE>)+ "]"
<RANGE> ::= (<RNGVAL> ":" <RNGVAL>)
<RNGVAL> ::= "$" | [0-9]+
```

*A single token can be provided also without a semicolon terminator

For identifiers, the modelica language specification has been followed, stating that “an identifier always start with a letter or underscore (_), followed by any number of letters, digits, or underscores.

To explain the syntax of an input string, an example is used

`“x1[1:2];y;mat[$:$,1:3];der(x1);”`

it’s meaning will be (information stored in the VF):

- print **x1**, because it’s a 1D array print only values for i = 1, i<2
- print variable with identifier **y**
- print **mat**, because it is a matrix print
  - first dimension: all values (**<u>$ means from the beginning or until the end</u>**)
  - second dimension: values for i = 1, i<3
- print derivative of variable x1 **der(x1)**

Keep in mind that if the model had another variable (with another name) it won't be printed because the variable filter is active and that identifier does not match any regex; the identifier it's not even specified as a tracked variable (see Variable Tracker).

For starters, in order to implement this parsing stage, a **Lexer** has been implemented. The Lexer returns tokens [0-255] if there is an unknown character, otherwise it returns one of the known tokens eof, der, der_close, tok_identifier, comma, colon and so on; token is an `Enum`. A series of handy global variable are used to keep track in the parsing of values such as the `variableIdentifier` that is being red, a boolean representing if an array is parsed, a pointer to the input string and several more.

A base class **ExprAST** has been introduced for all expression nodes: VariableExpr, ArrayRange, DerivativeExp, RegexExp with relative attributes, basically one for each "token type" that can be parsed.

**Grammar productions** are represented by functions that returns the corresponding AST node, and parsing is carried out by the method parseExpressionElement by eating the right tokens and calling the parsing methods. By doing so a Recursive Descent Parser has been built.

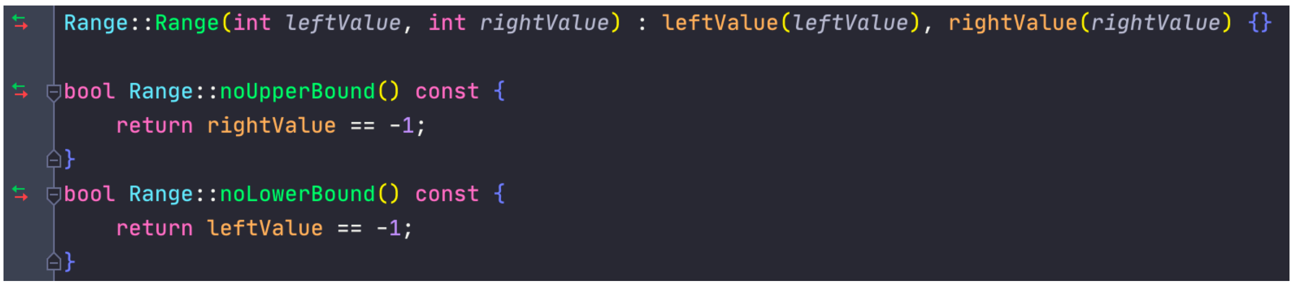
The **parsing result** is packed into a VariableTracker class, that will be further discussed, and registered into the VF.

After reading the string from command line arguments, the public method that parses and fills the variable filter is:

```
parser.parseCommandLine(<input string>, <variable filter reference>);
```

## 2.2 VariableTracker

A variable tracker is the basic object that can be registered into the variable filter, it represents a variable that must be printed (key is its identifier) together with different attributes. If it's specified as an array it is stored together with its dimensions and individual ranges. To represent **a range without a bound**, because range values are unsigned integers, the number -1 is used.



```
Range::Range(int leftValue, int rightValue) : leftValue(leftValue), rightValue(rightValue) {}

bool Range::noUpperBound() const {
    return rightValue == -1;
}
bool Range::noLowerBound() const {
    return leftValue == -1;
}
```

x[$:40] is stored with a `Range(-1, 40)`

## 2.3 VariableFilter

The variable filter class is the main representation of the filter, it stores **two unsorted tables** of VariableTrackers representing variables and derivatives and a list of regular expressions in the string form. Exposed methods are the following:

- `setBypass`: toggles bypass mode, variable filter behaviour disappear and everything is printed as before.

- `addVariable/Derivative`: two methods used by the parser to register a tracker into the current instance of the variable filter

- `addRegexString`: stores a string into the list of regexes, this string will be used later to build a regex object and match the model's variable identifiers

- `dump`: debug function that dumps / prints on `std::cout` a string representation of the variable filter state

- `matchesRegex/printDerivative/checkTrackedIdentifier`: boolean methods used to check if a provided identifier is tracked by the variable filter or not. Tracked means that is on the print list.

Because a variable tracker is uniquely identified by a variable name, lookups are commonly used and iterations not, an **hash table** has been chosen as a proper data structure to hold this values [Lookup **O(1)**]. There is no need to keep variables in alphabetical order so a **std::unordered_map** has been chosen.

For regexes given a string representing a variable identifier a check among all the stored regex must be performed, thus an std::list is used, easy to iterate within in.

Regular expressions are stored as strings but they are validated and matched by using **LLVM::Regex** class, it's easy to use and exposes the methods `match` and `isValid` that handle the needed tasks.
In order to force the regex to match the entire string a not a sub-string, make sure to add **'^'** in front of the regular expression and **'$'** at the end.

Lookup and boolean functions are used by code generation in order to establish if a specific value must be printed or not, lookup keys are variable declared in the model (.mo file). This means that if someone adds variable to be tracked with a non existing identifier, Marco wouldn't even look for it and thus it won't cause any error. Entries in hash table are added with `insert_or_assign` method, thus if two trackers with the same keys are added, the last one is tracked.

Next step after properly translating user provided strings into the corresponding trackers is to modify the code generation, that uses MLIR, in order to apply filtering. This process has been carried out with different steps, described in the next section.

---

## 3. Tweaking Code Generation

The goal of this sub task is extending and modifying one of the registered passes, especially the `SolveModel` pass in order to insert some new logic in the generation of the printf call, that it's going to print the values. Previosly Marco was printing all the values, now if the variable filter is not bypassed only the tracked elements have to be printed.

The target code as said before is inside the SolveModel pass, where a series of `OpRewritePattern` are triggered. A **RewritePattern** in MLIR is the common base class for all DAG to DAG replacements, this framework is widely used throughout MLIR for canonicalization, conversion, and general transformation. I've modified the **SimulationOpPattern**, that works on the <u>SimulationOp</u>. In MLIR everything is an operation, thus the simulation ops represents a modelica simulation and was already in charge of issuing a print function.

Given the insertion point, a `mlir::FuncOp` operation was created with name "print", it receives a struct loaded with all the values of the simulation. Via an ExtractOp operation it extracts them, then it generates a PrintOp with a vector of `mlir::Values`.

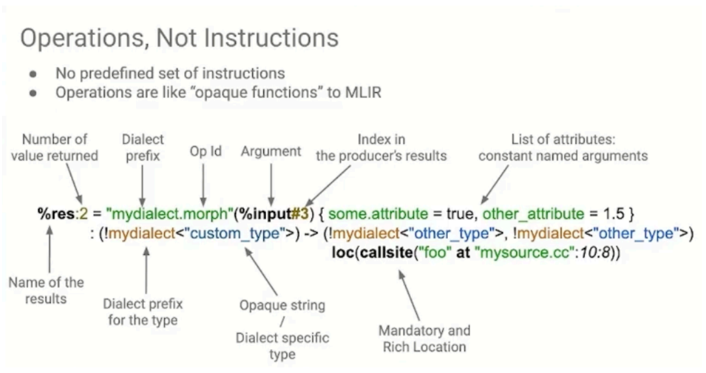
First step to start filtering out variables in this stage is having a vector containing all the variable names, this was not present in the SolveModel pass and it was one the first extension I made.

## 3.1 Carrying variable names in SolveModel pass

Why variable names (identifiers) are needed in this pass? The goal is to apply fine filtering, not only by name, also by derivatives, by array ranges and so on. The filtering must be deployed in this pass to be able to work with fine grain so names and needed information must be still reachable even at this level.

In order to have the names of model variables in this pass, and because the rewrite pattern works on the SimulationOp, this operation has been decorated with an **attribute** of type array, filled with variable names (`mlir::ArrayAttr variableNames`) that are later fetched and converted to a std::string vector in SolveModelPass: this tweak made names available in this pass.

To better understand what is an operation in MLIR and why I've used an attribute to store this kind of information, the picture below is provided:



Operations, Not Instructions

- No predefined set of instructions
- Operations are like "opaque functions" to MLIR

```
%res:2 = "mydialect.morph"(%input#3) { some.attribute = true, other_attribute = 1.5 }
         : (!mydialect<"custom_type">) -> (!mydialect<"other_type">, !mydialect<"other_type">)
                                           loc(callsite("foo" at "mysource.cc":10:8))
```

Number of value returned
Dialect prefix
Op Id
Argument
Index in the producer's results
List of attributes: constant named arguments
Name of the results
Dialect prefix for the type
Opaque string / Dialect specific type
Mandatory and Rich Location

<u>Prior to this modification</u> a print region* existed in the SimulationOp, in this region a vector of values to be printed was filled up <u>but only filtering by name was possible</u>. Because the goal is to move the print code generation to a lower pass, the SolveModel pass, this region was no longer useful and was **removed**. Operation traits have been subsequently updated due to the region removal. In MLIR **Traits** are a mechanism which

abstracts implementation details and properties that are common across many different attributes/operations/types/etc such as number of operands, number of regions, no return.

*(*) MLIR Regions are a list of basic blocks nested inside an Operation, basic blocks are a list of operations. As stated before everything is an operation in MLIR, operation can have a region, a region is a list of blocks, blocks are a list of operations.*

## 3.2 Filtering of Identifiers and Arrays

Now that all the names of the variables declared in the model.mo file are available in the SolveModel pass, the required filtering can be carried out. At this point the body of the print function is being written, a function that receives a struct as the only parameter containing the values from the simulation. For starters I've extracted values from the struct via **ExtractOp** into vectors

```
llvm::SmallVector<mlir::Value> valuesToBePrinted;
llvm::SmallVector<mlir::Value> derivativesValues;

/* Mlir Values: This class represents an instance of an SSA value in the MLIR
system, representing a computable value that has a type and a set of users. An
example of a value is the result of an MLIR operation. */
```

All values are extracted because filtering is performed in the next stage; the order of mlir::Values matter, it matches exactly the order in which variables are declared in the model .mo file,

e.g.

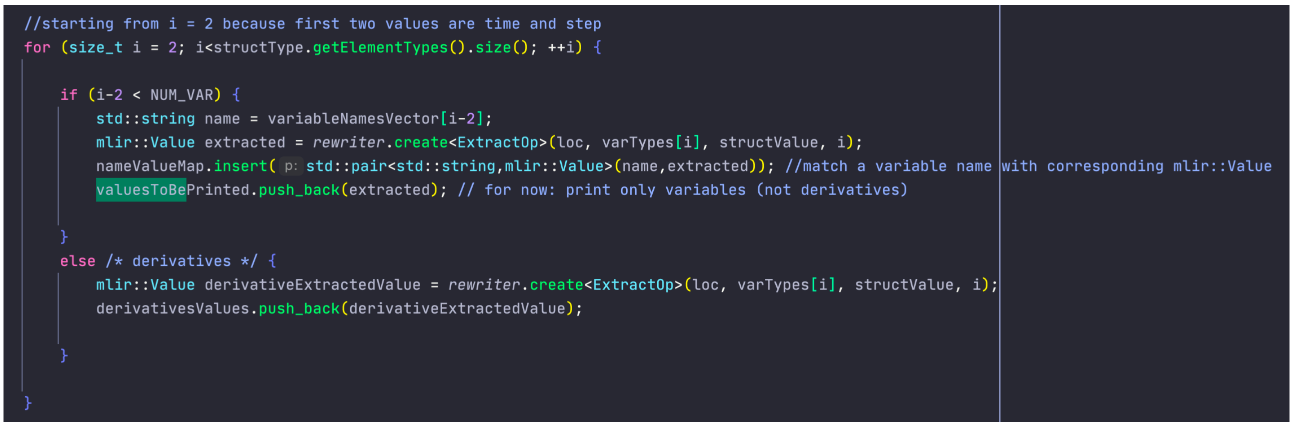
**Model:**
```
final parameter Real tau = 5.0;
Real[4] x1(start = 0.0);
Real y(start = 9.9),

… = der(x1[1]) * … //derivative is used
```

**Extracted values:**

```
time (always present) | step (always present) | tau | x1 | y || der(x) (then derivatives begin)
```

The code performing this task is the following:



```
//starting from i = 2 because first two values are time and step
for (size_t i = 2; i<structType.getElementTypes().size(); ++i) {

    if (i-2 < NUM_VAR) {
        std::string name = variableNamesVector[i-2];
        mlir::Value extracted = rewriter.create<ExtractOp>(loc, varTypes[i], structValue, i);
        nameValueMap.insert( p: std::pair<std::string,mlir::Value>(name,extracted)); //match a variable name with corresponding mlir::Value
        valuesToBePrinted.push_back(extracted); // for now: print only variables (not derivatives)

    }
    else /* derivatives */ {
        mlir::Value derivativeExtractedValue = rewriter.create<ExtractOp>(loc, varTypes[i], structValue, i);
        derivativesValues.push_back(derivativeExtractedValue);

    }

}
```

Number of variables, variable names, derivative (y/n), array dimensions and sizes are fixed at compile time and can be used in filtering, dynamic dimension can also be used. Values of course are computed at runtime.

Now that all the values have been extracted, without the variable filter they would be printed indistinctly. An iteration is performed among all values and print code is generated according to the variable filter current configuration. Iteration is performed first on the values of variable, later on derivatives, this is to keep a similar output to the one previous to my modification (time -> vars -> derivatives). The helper functions used generate the code that would invoke printf to print the values is the same used in other parts of marco, copied in this pass.

For each value, print generation code region is entered if the following logic condition is evaluated true:

```
variableFilter.isBypass() || regexMatched || variableFilter.checkTrackedIdentifier(varName)
```

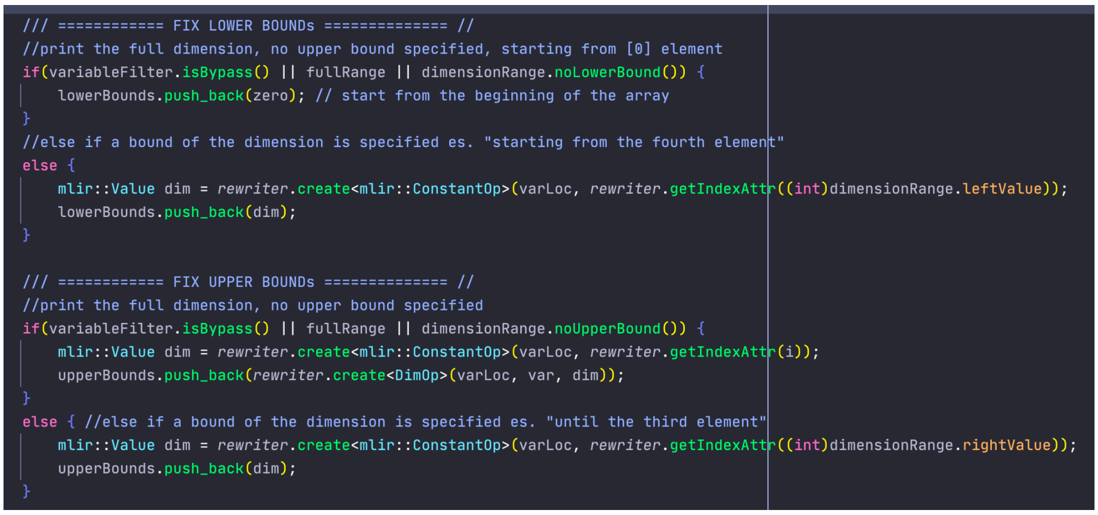
Which can be interpreted as:

*"print variable of name varName if the variable filter is not active, if its identifier matches one of the provided regex or if it has been specified to be printed (also valid with array)"*

Now that we know that (just by looking at its name) a variable have to be printed, if the variable it's an array a further filtering of each range (for each array dimension or rank) must take place.

The code to generate the code (and for loops) for printing out an array value is handled by **MLIR SCF dialect**, `mlir::scf::buildLoopNest` function creates a perfect nest of "for" loops. In order for this to work **3 objects** must be properly filled out:
- **step**: increment at each for iteration, it's always one in this case because array is scanned element by element // similar to C " i = i + step "
- **lowerBounds** vector: one element from each array value dimension, it represents the index at which each iteration starts // similar to C " i = INIT "
- **upperBounds** vector: one element from each array value dimension, it represents the upper bound of each iteration, similar to C " `while (i < UpperBound)`"

Ranges are validated (check that are within the size of the current array dimensions) and then are used to correctly bound each iteration. The code is the following, dimension is retrieved at runtime by emitting **DimOp**.
This is performed for each array dimension (rank).



```
/// ============ FIX LOWER BOUNDs ============== //
//print the full dimension, no upper bound specified, starting from [0] element
if(variableFilter.isBypass() || fullRange || dimensionRange.noLowerBound()) {
    lowerBounds.push_back(zero); // start from the beginning of the array
}
//else if a bound of the dimension is specified es. "starting from the fourth element"
else {
    mlir::Value dim = rewriter.create<mlir::ConstantOp>(varLoc, rewriter.getIndexAttr((int)dimensionRange.leftValue));
    lowerBounds.push_back(dim);
}

/// ============ FIX UPPER BOUNDs ============== //
//print the full dimension, no upper bound specified
if(variableFilter.isBypass() || fullRange || dimensionRange.noUpperBound()) {
    mlir::Value dim = rewriter.create<mlir::ConstantOp>(varLoc, rewriter.getIndexAttr(i));
    upperBounds.push_back(rewriter.create<DimOp>(varLoc, var, dim));
}
else { //else if a bound of the dimension is specified es. "until the third element"
    mlir::Value dim = rewriter.create<mlir::ConstantOp>(varLoc, rewriter.getIndexAttr((int)dimensionRange.rightValue));
    upperBounds.push_back(dim);
}
```

Now that bounds and step are correctly fixed according to the variable filter state, they are passed to the `buildLoopNest` function as parameters and code is emitted properly.

### 3.3 Filtering of Derivatives

One last filtering behaviour have to be implemented, derivatives. Someone must be able of print just “y” and not “der(y)”.

Information needed is a map that for each variable names tells which one has a derivative, an intermediate map between mlir::Values is used in order to determine if given a variable value it exists the corrisponding derivative Value.

This secondary map is already filled during another rewrite pattern, DerOp, where derivatives are substituted by allocations. A map <string, bool> is build based on this that links each variable name to a boolean (true if variable has a derivative).

After generating the code for the variable values, filtering is carried out with derivative values.

First of all because for each variable name I want to check if it has a derivative and it that case ‘pop’ the derivative value out of a data structure and send it to print, a **std::queue** is build with derivative values. Derivative values are in the same order of the corresponding variables so a **FIFO** fashion data structure is the one that makes this job easier, Pop is O(1).

Now that all these elements are there, filtering is trivial and can be summarised by the following sentence:
For each variable name (declared in the model) check if it has a derivative Value associated, then if VF is bypassed or der(variable) is on the print list generate the code to fully print it.

---

## 4. Usage

When running marco, command line arguments must be used in order to send the specification string to the variable filter.

Existing command line arguments have been extended with

```
static cl::opt<string> variableFilterString("vf", cl::desc("<variable-filter-string>"),
cl::init("bypass"), cl::value_desc("vfstring"));
```

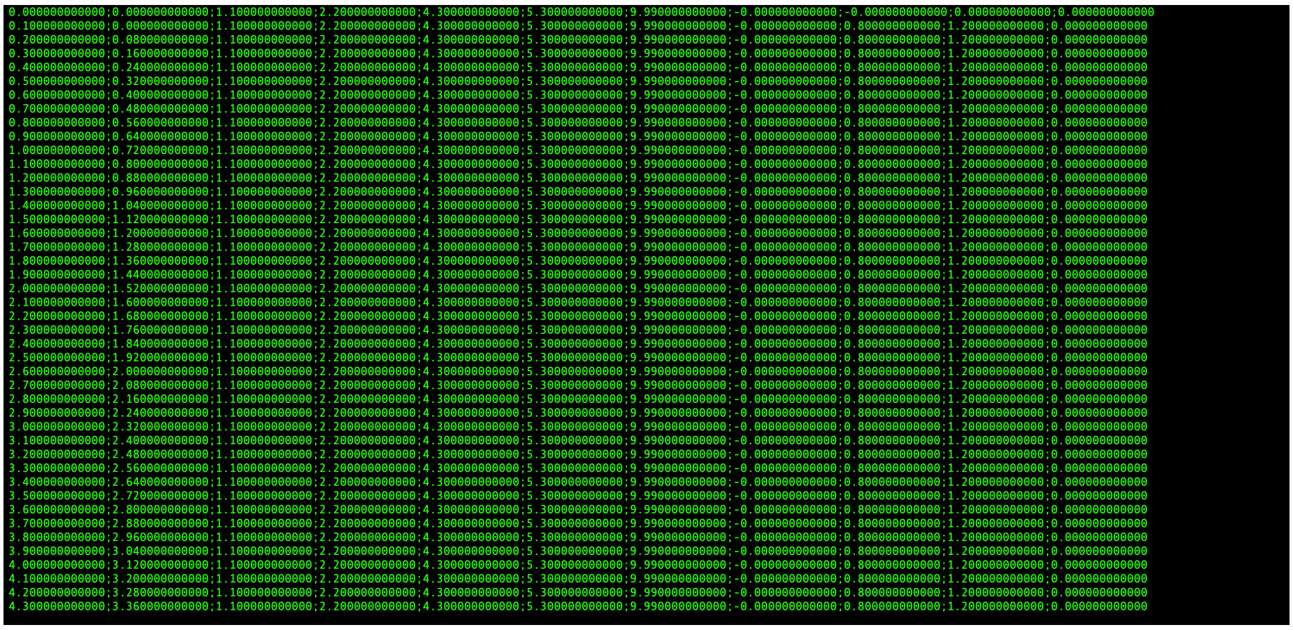
This way a string can be passed to marco via command line arguments e.g.
**`marco /marco/model.mo -vf "x1[1:2];y;mat[$:$,1:3];der(x1);"`**

Variable filter becomes active after adding at least 1 variable or derivative, absence of -vf will bypass the variable filter.

A test .cpp file has been added to utils's test folder to carry out basic unit testing of all the VariableFilter related classes. Tests of parsing and correct tracker storing for variable identifiers, arrays, ranges, regexes and derivatives. Numerous assertions are used to check if the variable filter state changes accordingly to parsed token.

For the code generation part, generated intermediate representation has been compared in order to solve bugs and check if the loop iterations were running between the right bounds. Visual print output has been compared and tested with expected one in order to check if filtering was applied properly. Bypass, regexes, identifiers and array slice ranges have been tested and compared with filter active or bypassed. An example is following:

```
model SimpleDer
  final parameter Real tau = 5.0;
  Real[4] x1(start = 0.0);
  Real[2,3] mat ={ {1.0, 1.1, 2.2},{3.3, 4.3, 5.3}};
  Real y(start=9.99);
equation
  tau*der(x1[1]) = 1.0;
  2.0*der(y) = tau;
  for i in 2:4 loop
        tau*der(x1[i]) = 2.0*i;
  end for;
end SimpleDer;
Variable filter configuration is given by: "x1[1:2];y;mat[$:$,1:3];der(x1);"
Printed values are correctly printed as: TIME | X1[1] | Y | MAT[ALL][1 and 2]; DER(X1)
```



```
0.000000000000;0.000000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;-0.000000000000;0.000000000000;0.000000000000
0.100000000000;0.000000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.200000000000;0.080000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.300000000000;0.160000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.400000000000;0.240000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.500000000000;0.320000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.600000000000;0.400000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.700000000000;0.480000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.800000000000;0.560000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
0.900000000000;0.640000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.000000000000;0.720000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.100000000000;0.800000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.200000000000;0.880000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.300000000000;0.960000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.400000000000;1.040000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.500000000000;1.120000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.600000000000;1.200000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.700000000000;1.280000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.800000000000;1.360000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
1.900000000000;1.440000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.000000000000;1.520000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.100000000000;1.600000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.200000000000;1.680000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.300000000000;1.760000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.400000000000;1.840000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.500000000000;1.920000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.600000000000;2.000000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.700000000000;2.080000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.800000000000;2.160000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
2.900000000000;2.240000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.000000000000;2.320000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.100000000000;2.400000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.200000000000;2.480000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.300000000000;2.560000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.400000000000;2.640000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.500000000000;2.720000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.600000000000;2.800000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.700000000000;2.880000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.800000000000;2.960000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
3.900000000000;3.040000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
4.000000000000;3.120000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
4.100000000000;3.200000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
4.200000000000;3.280000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
4.300000000000;3.360000000000;1.100000000000;2.200000000000;4.300000000000;5.300000000000;9.990000000000;-0.000000000000;0.800000000000;1.200000000000;0.000000000000
```

## 6. Conclusions

Modelica is widely used between engineers and this project has been an unique opportunity for me to apply my C++ knowledge in a new environment and learn more about compilers. After learning the basics of the Modelica language I begun to implement the parsing and input processing step, which does not involve MLIR. Then, after building LLVM from scratch, I've been learning MLIR and the LLVM framework through examples, documentation and Google developer conferences.

In order to develop the code generation part of the project, basic knowledge and reasoning techniques about writing compilers have been successfully learnt in tutoring session by D. Cattaneo of Formal Languages and Compilers course at Polimi.

Roadmap and implementation hints have been given by graduate student M. Scuttari.

Using and understanding MLIR has non been easy but I've understood the flexibility of the framework and its importance in Google TensorFlow.

This projects makes the output filterable by cutting out unnecessary stuff, lower compiler passes now have more informations on model variables and derivatives.

Further works can bring more a better output form, a GUI to filter out variables in a handy way and to overlay different graphs.