Efficient Compilation of Large Scale Dynamical Systems

Federico Bergero[1,2] Mariano Botta[2] Esteban Campostrini[2] Ernesto Kofman[1,2]

[1]CIFASIS, CONICET, Argentina {bergero,kofman}@cifasis-conicet.gov.ar
[2]FCEIA, UNR, Argentina {marianoabotta,lesteban22}@gmail.com

Abstract

In this work, we present a novel methodology to efficiently compile large scale dynamical systems described as Modelica models, and its implementation in a prototype Modelica Compiler called ModelicaCC. The methodology allows to perform the different stages of the compilation process without expanding the content of repetitive structures so the resources (CPU time and memory) used by the compiler result independent on the model size. Besides introducing the methodology with their algorithms and the implementation in the ModelicaCC compiler, we analyze their efficiency comparing its performance with that of OpenModelica in different large scale models.
Keywords: Modelica Compilers, Large Scale Models, Tarjan Algorithm, Model Flattening

1 Introduction

Modelica (Fritzson, 2004) is an object-oriented, equation-based language for representing continuous and hybrid models. Modelica provides a standardized way to model complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power, or process-oriented subcomponents.

The simulation of these models requires some transformations. First, classes and connections amongst them are removed obtaining a *flat* model. A flat model yields a Differential Algebraic Equation (DAE) system which must be then sorted and converted into an Ordinary Differential Equation (ODE) system. From the ODE representation, C code is generated and compiled together with the ODE numerical integration method. This pipeline is performed by the Modelica compilers.

Frequently, Engineers and researchers of different domains need to simulate large scale models, which are usually the result of connecting together several identical components in repetitive structures. Examples of these models appear in Smart Grids, spatial discretization of Partial Differential Equations (PDE), etc. Although the Modelica language do allow to represent these large scale models in a convenient way, Modelica compilers fail to complete the compilation pipeline when the size of the models grows beyond a few thousand of components. Thus, efficient handling of large scale models is an important topic in the modeling and simulation community (Cellier et al., 2013).

In this article we present novel algorithms to address the compilation of large scale Modelica models. The idea behind them is to exploit the repetitive nature of large scale models and perform all the mentioned transformations (flattening, sorting and code generation) without expanding the model arrays. Also, as outlined in (Stavaker, 2011), preserving the iterative equations until the code generation phase enables the use of parallel simulation techniques that would otherwise be useless.

We present also prototype implementations for these algorithms and compare their performance against OpenModelica compiler.

The work is organized as follows: Section 2 provides the main concepts used along the rest of the article. Then, Sections 3 and 4 introduce the novel algorithms developed for the flattening and causalization stages. After that, Section 5 presents the ModelicaCC compiler, describing its architecture and components. Finally, a comparative study of the Compiler performance on large scale systems is performed in Section 6 and the conclusions are presented in Section 7.

2 Background

In this section we first describe the process for simulation of Modelica models and we outline the problems faced when compiling large scale systems. Later we present the tool we use for generating C code and simulating, and finally we review some works related to this article.

2.1 Modelica Compilers

As mentioned earlier, Modelica models require different transformations before being simulated. First, a flattening stage is responsible for converting the Modelica model into an equivalent model without classes (inheritance and composition), converting also the `connect` equations into equalities. After this stage, the **flat** model only contains variables of basic types (real, integer, boolean, etc) and equations (and algorithms) representing a hybrid DAE system.

This DAE system is then converted into an ODE system so it can be simulated by ODE solvers (Runge-Kutta, DASSL, DOPRI, etc). This conversion is divided in two sub-steps. First, if the problem has high index, an index reduction algorithm (such as Pantelides (Pantelides, 1988)) is applied. Then, the equations are horizontally and vertically sorted by a matching algorithm (such as Tarjan (Tarjan, 1972)) and the ODE system is obtained.

Then, an optimization stage is applied removing trivial equations (like `a = b`). Finally, C language code is generated and compiled together with the numerical solver obtaining an executable program that simulates the model.

Figure 1 shows a typical pipeline found on most Modelica tools, like OpenModelica (Fritzson et al., 2005), Dymola (Brück et al., 2002) and others.

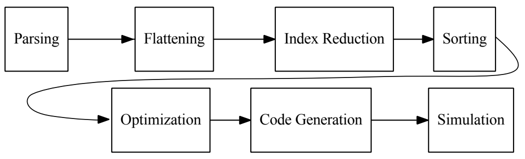

Figure 1. Pipeline of compilation

2.2 Modelica and Large Scale Models

With the availability of more powerful simulation platforms, models that in the past were dozen or hundreds of variable are increasing their size to thousands or millions of variables (Cellier et al., 2013). This imposes some new challenges on the modeling and simulation community. First we must have ways of modeling these huge systems. Modelica seems to be a good choice for that. By their nature, large scale models are rarely developed by extension, i.e. they are not handwritten. In general they possess some repetitive structure that makes them easier to be described by comprehension.

Modelica is well fitted for this kind of description. It allows the modeler to replicate components and connect them in a regular fashion. Thus a large scale model can be written with a short Modelica description. This is usually done using the `for` iterative equation for behavior description and array variables for states.

However, problems appear in the different stages of the compilation pipeline. When flattening large scale models, most Modelica tools perform what is known as loop unrolling, i.e. `for` equations are replaced by their equivalent scalar equations. In this stage array variables are also expanded into several scalar variables.

Consider for instance the lumped model of a LC Transmission Line depicted in Fig.2. This model can be described by the Modelica code of Listing 1.

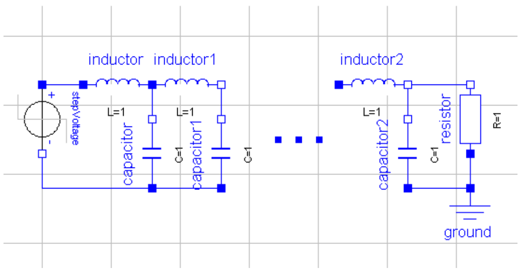

Figure 2. LC Line model

Listing 1. Hierarchical Modelica model of the LC Line example

```
model lcline
  import Analog = Modelica.Electrical.Analog;
  model lcsection
    Analog.Interfaces.Pin pin2;
    Analog.Basic.Inductor i;
    Analog.Interfaces.Pin pin;
    Analog.Interfaces.Pin pin1;
    Analog.Basic.Capacitor c;
  equation
    connect(c.p, pin1);
    connect(pin, i.p);
    connect(i.n, c.p);
    connect(c.n, pin2);
  end lcsection;

  Analog.Basic.Resistor r;
  Analog.Basic.Ground g;
  constant Integer N = 100;
  lcsection[N] lc;
  Analog.Sources.ConstantVoltage s;
equation
  connect(r.n, g.p);
  connect(s.n, g.p);
  connect(s.p, lc[1].pin);
  connect(r.p, lc[N].pin1);
  connect(g.p, lc[N].pin2);
  for i in 1:N - 1 loop
    connect(g.p, lc[i].pin2);
    connect(lc[i].pin1, lc[i + 1].pin);
  end for;
end lcline;
```

If we ask OpenModelica to flatten this model, the following code is obtained:

Listing 2. Flat model without arrays

```
class lcline
  constant Integer Size = 100;
  Real lc[1].c.v;   Real lc[1].c.i;
  parameter Real lc[1].c.C;
  ...
  Real lc[100].c.v; Real lc[100].c.i;
  parameter Real lc[100].c.C;
  ...
equation
   lc[1].c.i = lc[1].c.C * der(lc[1].c.v);
  ...
   lc[100].c.i = lc[100].c.C * der(lc[100].c.v);
  ...
end lcline;
```

In this code, we can see several equations like `lc[1].c.i=lc[1].c.C*der(lc[1].c.v)`, `lc[2].c.i=lc[2].c.C*der(lc[2].c.v)`, etc. coming from the `Capacitor` class inside each instance of the `lcsection` model.

That way, the cost of flattening this model is (at least) linear w.r.t. the `Size` parameter. Moreover, this expanded version will impose a huge burden on the successive stages of the compilation process as they must now deal with a large description.

In this work we argue that for efficient compilation of large scale models, all the steps involved in the process must be performed without expanding the model at all. Therefore in each step we deal with a large scale model but described in a short Modelica code.

2.3 μ–Modelica and QSS Stand–Alone Solver

The QSS Stand–Alone Solver (Fernández and Kofman, 2014) is an open source tool for continuous system simulation. The solver has a complete implementation of the QSS methods (Cellier and Kofman, 2006) and it includes also DOPRI and DASSL (Petzold, 1983) solvers.

The models must be described as a hybrid ODE system using a subset of the Modelica language called μ–Modelica. μ–Modelica contains the basic Modelica statements that allow to represent hybrid ODE systems as they are used by numerical ODE solvers, but using Modelica syntax instead of C or FORTRAN language. That way, the μ–Modelica code is easily translated by the QSS solver into C code and compiled together with the numerical integration method of choice.

A remarkable feature of this tool is that it does not unroll `for` loops in the generated C code. This has two upsides, first the computational cost of the translation does not depend on the size of the loop (hence the size of the model), second this also holds for the compilation of the generated C code.

ModelicaCC compiler uses the QSS solver for the last stages of the compilation process (C code generation and simulation). Thus, the goal of the previous stages is to translate a Modelica model into μ–Modelica.

2.4 Related Work

Some work has been done regarding the compilation of large scale Modelica models. Studies have been developed testing how compilers work on large scale models recognizing their limitations on that area (Frenkel et al., 2011; Sezginer, 2014-2015; Casella, 2015). In (Jens Frenkel, 2012) particularly, the authors study different causalization (matching) algorithms applied to large scale models and conclude that the PF+ algorithm (by Duff) is the best choice as it achieves linear performance on the tested cases.

In (Zimmer, 2009) Zimmer also presents the problem compiling large scale models of current Modelica compilers. There the author proposes a solution based on the preservation of module structure during the compilation phase. This is achieved through partial flattening and causalization.

More closely related work was presented in (Arzt et al., 2014), where the authors explore the idea of finding repetitive structure on the incidence graph to efficiently apply Pantelides index reduction algorithm without dealing with flattening nor causalization. Therefore both works are complementary. The present work relates to this article in the sense that we also try to exploit the repetitive structures, but in our case, we develop new algorithms (in Section 3 and 4) to be applied on the an extended graph.

A first attempt to preserve array variables and iterative equations was presented in (Stavaker et al., 2010; Stavaker, 2011). There, the authors modified the flattening and causalization algorithm of OpenModelica to allow slice variables (such as `a[1:10]`) to be treated as a single variable. The modified algorithm does not correctly flattens hierarchical models and does not handle irregular definition of variables. As we will see, our proposal deals with more general and realistic cases (both while flattening and sorting).

3 Flattening Algorithm

This section presents an algorithm to efficiently flatten large scale models, without expanding equations and array variables. This stage is divided in two steps. The first one actually flattens the model and the second one removes the connect statements by the corresponding equations.

3.1 Class Flattening Stage

This stage of the flattening algorithm deals with class flattening leaving connections (and connectors) untouched. The algorithm follows the principles of most Modelica compilers, but with a special treatment for arrays of variables.

For example, when flattening the model shown in Listing 1 most Modelica tools will expand the array `lc.c.v` into `Size` variables and generate `Size` equations of the form `lc[1].c.i=lc[1].c.C*der(lc[1].c.v)`. While the result is correct, it will produce a large description for the successive stages of the compilation pipeline.

What we propose is to avoid expanding the arrays to generate the flat model. In the transmission line example of Listing 1, this can be done as follows:

- First flatten model `lcsection` by flattening their components `Capacitor` and `Inductor`. This will result in a model with basic type variables.

- For flattening the array `lcsection lc[Size]` we proceed as follows:
  - For each variable in the flat model `lcsection` we define an array of size `Size`, prefixing the name of that variable with `lc_`.
  - We change the modifications (if any) on the new array variable.
  - We wrap each equation in a `for` loop (with range `i in [1:Size]`) adding the `[i]` index expression on each use of the above defined variables. The same is done for algorithmic sections.
- Now we are ready to remove the `lc` instance of `lcline`.

The result of applying these rules is shown in Listing 3.

Listing 3. Flat model without expanding arrays

```
model lcline
  constant Integer Size=100;
  parameter Real lc_c_C[Size];
  Real lc_c_v[Size];
  Real lc_c_i[Size];
  ...
equation
  ...
  for i in 1:Size loop
    lc_c_i[i] = lc_c_C[i]*der(lc_c_v[i]);
  end for;
end lcline;
```

We see that the length of this flat version is independent of the `Size` parameter.

The idea sketched in this example can be applied to most arrays of classes. However, if the array has a non-regular definition (due to the usage of a *type redeclare* modification in some components of the array, for instance), then a special procedure should be followed. Anyway, those cases are beyond the scope of this work since most practical large scale models do show regularity in their class definitions.

3.2 Connection Replacement Stage

After the class flattening stage, we still have to remove connectors and convert the `connect` operators into equations. This process cannot be done locally (at the component level) since connections are inter–component operations. Thus, we have no choice but to solve this problem looking at the complete model.

A connect equation binds the variables of two connectors instances. Modelica distinguishes between two kind of variables inside connectors:

Potential variables which are equalized for connected connectors.

Flow variables which are zero-summed for connected connectors and zero-equalized for unconnected ones.

When replacing a `connect` operator, the potential variables are easily handled (they are converted to an equation like `a=b`). However, flow variables require a special treatment since more than two connectors can be connected together resulting in a zero-sum of multiple terms. To generate these equations we must compute the set of all connectors that share a connection.

We can associate this issue to a graph theory problem as follows. First we build an undirected bipartite graph with one node for each `connect` operator and one node for each connector instance. Then, for each connect node, we add two edges linking it with the corresponding connector nodes. Then, computing the connection sets is analogous to finding the connected components of the graph. Several search algorithms, like Depth First Search (DFS) (Hopcroft and Tarjan, 1973), achieve this goal with linear complexity.

Connections on Large Scale Models After the class flattening stage, arrays are preserved appearing inside iterative equations arising from scalar equations wrapped in `for` loops. As mentioned above, the `connect` equations and connector variables are still part of the model and they should be replaced by their equivalent equations.

If these connections and connectors belong to replicated models, our purpose is to replace them by the corresponding equations preserving the arrays and the `for` loops. To this end, we propose to extend the graph theory procedure explained above representing arrays of connectors and connections by single nodes (with some additional information) in order to find the connected components on this *vectorized* graph.

3.2.1 Vectorized Connection Graph

We build the vectorized connection graph as follows

- For each `connect` operator we create an e node. Each operator inside a `for` loop counts as a single node.
- For each connector, we add a c node. An array of connectors also counts as a single node.
- We add two edges for each `connect` operator linking it with the two connectors involved. Each edge will have the range $P \subset \mathbb{N}$ of use of each connector (in the case it corresponds to an array) as a property.

3.2.2 Vectorized Connection Algorithm

Now we must find the connected components in the vectorized graph. A modified DFS algorithm is proposed for that goal. The idea is to gather multiple connected components while traversing the graph using the information available on the edge properties.

Before presenting the algorithm, we provide the following definitions:

Definition 1 *Given a vectorized graph $\mathscr{G}$ and a range $P_0 \subseteq \mathbb{N}$, we define $\mathscr{G}(P_0)$ as the subgraph resulting containing all the nodes of $\mathscr{G}$ and only the edges with range P such that $P_0 \subseteq P$.*

Definition 2 *Given a vectorized graph $\mathscr{G}$, we say that a subset $\mathscr{C}$ of its nodes is connected in the range P_0 if it is connected in the classic sense on graph $\mathscr{G}(P_0)$.*

Definition 3 *Given a vectorized graph $\mathscr{G}$, a node c, and a range P_0, we say that the set of nodes $\mathscr{C}$ is a connected component including c in the range P_0 if it is a connected component including c in the classic sense on graph $\mathscr{G}(P_0)$. We shall denote it $\mathscr{C} = CC(\mathscr{G}, c, P_0)$.*

Definition 4 *Given a range P_0 and a node c of a vectorized graph $\mathscr{G}$, we say that P_0 is a compact range of c if $CC(\mathscr{G}, c, P_0) = CC(\mathscr{G}, c, P)$ for all $P \subseteq P_0$.*

Notice that if P_0 is a compact range of c, then the connected component (cc) is the same for each subrange of P_0. This means that all the variables involved in the cc have the same connection structure in the range P_0. Thus, they can be treated in the same way.

Thus, given a vectorized graph, finding a large compact ranges allows to gather multiple components on a single step. The following algorithm performs this step.

Given a node c, we initially take P_0 as the range of one of its edges, and then:

1. Remove all the edges whose range P has empty intersection with P_0.
2. In the resulting graph, find the classic connected component $\mathscr{C}$ ignoring the edge ranges.
3. If two nodes of the connected component $\mathscr{C}$ contain an edge with range P such that $P \cap P_0 \neq P_0$ then P_0 is not a compact range. Thus, take $P'_0 = P \cap P_0$ and go back to step 1.
4. Otherwise P_0 is a compact range including node c of the vectorized graph. Moreover, $\mathscr{C} = CC(\mathscr{G}, c, P_0)$ is the connected component including c in the range P_0.

Finally, the previous algorithm can be iteratively used to find all the connected components on compact ranges on a vectorized graph $\mathscr{G}$ as follows.

1. Take a node c with at least one edge with non empty range. If none is found, all the connected components have been already computed.
2. Compute a compact range P_0 and the corresponding connected component $\mathscr{C} = CC(\mathscr{G}, c, P_0)$ for node c using the previous algorithm.
3. Add $\mathscr{C}$ and P_0 as a new connected component to the result.
4. Remove the range P_0 from all the edges in $\mathscr{C}$ and go back to step 1.

This algorithm provides a set of connected components with the corresponding ranges. For instance, in the model of Listing 1 one of the connected components will be:

$$\{g.p, lc[1:Size].pin2, r.n, s.n\} \tag{1}$$

The presented algorithm is not linear as the original DFS since it visits many times the same node. Anyway, the fact that we have only one node per array and one node for each iterative equation makes this algorithm significantly faster than its scalar counterpart in large scale models. Additionally, it allows to generate equations preserving the arrays and `for` loop equations.

For simplicity and space reasons, we only introduced the basic algorithm which does not cover all cases. Scalar variables inside loops have a special treatment, so that the edges have their ranges covering the whole set $\mathbb{N}$. Also, connections binding arrays with different ranges also need a special treatment which involves translating the ranges while traversing the vectorized graph.

Equation Generation Once we have computed the connected components we must generate their equations. Given a connected component we do:

- For each potential variable in the connector we add an equality equation binding their value. In the case of ranged connectors we include a `for` equation.
- For each flow variable we add a zero-sum equation. In the case with range connectors we include a `sum` term for all the elements involved.

For example, the first connected component of Listing 1 would yield the following equations:

Listing 4. Iteartive equations for solved components

```
// Potential variables
for i in 1:Size
  g_p_v = lc_pin2_v[i];
end for;
g_p_v = r_n_v;
g_p_v = s_n_v;
// Flow variables
g_p_i + sum(lc_pin2_i) + r_n_i + s_n_i= 0 ;
```

4 Causalization Algorithm

This section presents a novel algorithm to convert a DAE system into an ODE system specially tailored for large scale models. This procedure can be also expressed as a graph theory problem. The idea is to capture the relation between equations and variables as an undirected bipartite graph and then applying Tarjan's algorithm to find the strongly connected components. We will review a simplified version of this algorithm presented in (Cellier and Kofman, 2006).

4.1 Classical Tarjan's Algorithm

We start from a flat Modelica model with N equations and N unknown variables [1].

Graph Creation

- For each equation we add a vertex e to the graph.
- For each unknown variable we add a vertex v to the graph.
- We add an edge (e ,v) if unknown v is used in equation e.

We will assume that unknown variables in Modelica models are of type `Real`. Parameters, constants and discrete variables are not considered unknown for the continuous part of the system. Also, assuming the absence of singularities, variables that appear inside a `der` operator are considered state variables which are known (with the unknown being their derivatives).

Causalization The causalization algorithm proceeds as follows:

1. Each e vertex of degree 1 (i.e., having only one outgoing edge) can be made causal since that equation has only one variable. Number the equation with the lowest available number starting from 1, follow the edge to its corresponding v node and remove all edges connected to v. Finally remove vertexes e and v.
2. Each v vertex of degree 1 can be made causal since that variable appears in only one equation. Then number the vertex with the highest available number starting from N, follow the edge to its corresponding e node and remove all edges connected to e. Finally remove vertexes e and v.
3. If we have numbered all equations we have finished. Else go to step 1.

[1]If the number of equations is different than the number of unkwonws the problem cannot be converted to an ODE system.

If a vertex (e or v) with degree 1 is not found, then an algebraic loop or a higher order singularity might exist, and a different procedure should be followed.

The space complexity of the algorithm is $\mathcal{O}(E+V)$ (with V number of vertex and E number of edges). The time complexity is also linear (if care is taken to find vertexes of degree 1) since each step removes one pair of vertexes.

When the algorithm finishes, the results is a sorted list (sorted by the number we assigned during the algorithm) of pairs (e,v) meaning that variable v must be solved using that equation e. Finally we solve each variable in each equation and we obtain a model in an ODE form.

Tarjan on Large Scale Models In order to apply Tarjan's algorithm to a large scale model, we should first unroll the `for` equations. This step alone takes a computational cost (time and space) proportional to the number of equations inside the `for` loop. Then, building the bipartite graph requires to create one e vertex for each unrolled equation and one v vertex for each element in the array variables (since `a[1]` and `a[2]` are different unknowns) as well as the edges of the graph. Finally, the cost of applying Tarjan's algorithm to this graph grows linearly with the number of vertexes.

In order to avoid this, we propose a modified Tarjan algorithm that is applied directly on the non-expanded model. This way, we not only avoid the cost of the expansion but this also results in applying Tarjan's algorithm to a much smaller graph and producing a much shorter ODE system description.

4.2 Vectorized Graph Representation

Here we start with a model with N_E (scalar or `for`) equations and N_V (scalar or array) variables. We will assume that every `for` loop has a single equation in its body. If this is not the case, it can be split into multiple `for` loops with only one equation in their body.

Then, we build an augmented bipartite graph where each edge has two properties: an equation range $p_e \subset \mathbb{N}$ and an index range $p_v \subset \mathbb{N}$ (two sets of integer numbers).

The graph is built as follows:

- We create an e vertex for each equation. If an equation is inside a `for` loop, only one node is created.
- We create a v vertex for each variable. Arrays are treated as a single variable.
- We add an edge (e,v) if unknown v is used in equation e with their p_e, p_v properties computed as follows:
  - $p_e = \{1\}$ if the equation is not inside a `for` loop.
  - Otherwise, p_e is the range of the `for` loop.

- $p_v = \{1\}$ if the variable is a scalar.
- Otherwise, p_v is the set of indexes of the array that are used at equation e.

For example the equation

```
for i in 2:10 loop
  der(a[i]) = b[i-1];
end for;
```

would have two edges, one connecting to array `der(a)` with properties $p_e = 2:10, p_v = 2:10$ and another edge connecting the equation with the array `b` with properties $p_e = 2:10, p_v = 1:9$.

If we compare this graph with the one resulting of expanding arrays and loops, we can see that the vectorized graph collapses all the variable vertexes of the same array into a single macro-vertex and all the equation vertexes of a same loop into a single macro-vertex. Also, edges have been merged so that a multi-edge with $p_e = [2:10]$ is equivalent to nine simple edges.

We propose next a Vectorized Tarjan's Algorithm to be applied to these augmented graphs.

4.3 Vectorized Tarjan's Algorithm

A vectorized version of the Tarjan's algorithm explained above can be sketched as follows:

1. Each e vertex of degree 1 can be made causal since those equations have only one variable each. Number the equation with the lowest available number starting from 1, follow the edge with properties p_e^1, p_v^1 to its corresponding v node and remove p_v^1 from the index range p_v of every outgoing edge connected to v. Remove edges with empty index range ($p_v = \{\}$), remove vertex e, and finally remove v if it has no more edges.

2. Each v vertex of degree 1 can be made causal since those variables are used in only one equation each. Number the equation with the highest available number starting from N, follow the edge with properties p_e^N, p_v^N to its corresponding e node and remove p_e^N from the equation range p_e of every outgoing edge connected to e. Remove edges with empty equation range ($p_e = \{\}$), remove vertex v, and finally remove e if it has no more edges.

3. If the graph is now empty, we have finished. Otherwise, go back to step 1.

In rule 2, N is the number of scalar variables we would have if we expand all arrays. The reason is that there are cases in which the algorithm above may actually expand some or even all the arrays.

Simple scalar cases Let us analyze first how this algorithm works for models without arrays and iterative equations. Here, the vectorized graph will have one e vertex for each equation and one v for each variable and all the edges will have $p_e = p_v = \{1\}$. When we make causal a vertex (e or v) we must follow the corresponding edge and compute the set difference of p_e^1 or p_v^1 with those of all outgoing edges. That difference will always be the empty set since all edges have the same p_e, p_v thus we will always remove all edges. Therefore, the Vectorized Tarjan's Algorithm fails back to the classical algorithm for simple scalar models.

Vectorized cases Let us see what happens on models with arrays and iterative equations on the two rules we have.

In rule 1, we make causal an e vertex using an edge with $p_v = \{i_1, i_2, \ldots, i_m\}$. That multi-edge represents m simple edges meaning that equation e involves that set of indexes of the array `v` associated with the v node. When we causalize e, we are making causal variables `v[i1], v[i2], ...` in a **single step**. Once those indexes are causalized, they become known variables to the remaining equations and we remove them from the index ranges p_v of the remaining outgoing edges of v.

The same analysis can be performed regarding rule 2.

Non Covered Cases The Vectorized Tarjan's Algorithm can fail to find a vertex with degree 1. The reasons here are the same as in the classical Tarjan's algorithm, either the model is structurally singular or the model has an algebraic loop. In the present work we do not handle this type of problems with the Vectorized algorithm. If this case is found, appropriate algorithms can be applied (such as Pantelides) on the expanded model.

The algorithm presented can be extended to deal with certain structures that, without having algebraic loops, result in graphs where all nodes have degree larger than 1. For the sake of simplicity we have not address this in the present work, but the algorithm can be easily extended to cover the case where a node has degree 2 or higher but there are some indexes (in p_e or p_v) that are only used in one outgoing edge.

Anyway, the presented algorithm is still able to sort many practical models, as we shall illustrate in Section 6.

Complexity and Result Analysis The complexity analysis in this case is not as simple as in the classical Tarjan's. The space complexity is now $\mathcal{O}(N_E + N_V)$ since the graph has that number of nodes. Here we are assuming that the properties of the edges are stored not by extension but by comprehension (since they are continuous range of integers).

Each step causalizes at least one variable (or one element in the case of arrays), so in the worst case the time complexity is the same as in the classical Tarjan's algorithm. On models showing regular structure (as in large scale models), the algorithm will causalize more than one variable in each step reducing the time complexity. Going further, additional conditions can be imposed on the model to assure that algorithm achieves a $\mathscr{O}(1)$ time complexity w.r.t. the model size.

Equation Generation When the algorithm finishes, the result is a sorted list of pairs (e, v) where each pair has the two ranges p_e, p_v of the edge used to causalize it. From here, Modelica code can be generated with the sorted and solved equations. An equation with range larger than one will be placed inside `for` loops in the range p_e, so that variable `v[i]` is computed there for `i` in the range p_v.

5 The Modelica C Compiler

We have developed a Modelica C Compiler (or ModelicaCC) to test the algorithms presented above. The architecture of the compiler follows the usual pipeline of Figure 1. In this case, we decided that the input/output of each stage are valid Modelica models (except for the very last stage which produce C code).

Each stage converts the Modelica model fed as input into a "simpler" equivalent one. As we will be using the QSS Stand–Alone Solver (Section 2.3) for the C code generation and simulation, the goal of the previous stages is to obtain a valid μ–Modelica model.

One design goal of ModelicaCC is to reuse as much available code as possible. So, several open source libraries were used as part of the implementation. The C++ STL and Boost library were used for representing the AST and the Boost-Spirit library was used for parsing. Also, the GiNaC library was used for symbolic manipulation of equations and the Newton iteration implementation of the GSL library is used for solving algebraic loops. Graph algorithms were implemented with the Boost Graph Library.

Below, we shall briefly describe the stages of the ModelicaCC tool outlining the novel features.

5.1 Flattening Stage

The flattening stage of ModelicaCC implements the algorithm presented in Section 3.1 and 3.2. The input to this stage is a total hierarchical Modelica model. The transformation of the first stage are mainly implemented as AST Visitors using the Boost library. The algorithm presented in Section 3.2 is also implemented using the Graph Library from Boost.

The command `./flatter` performs this stage, producing a flat Modelica valid code, which in large scale models preserves arrays and `for` loops.

5.2 Alias Elimination

Usually, Modelica models contain hundreds of trivial equations of the form `a = b`, most of them coming from `connect` operators. In order to simplify the tasks of the successive stages (causalization and code generation), these alias variables are removed together with their binding equations, replacing then the removed variable by their corresponding alias.

In the ModelicaCC implementation we also remove array alias. For instance, an equation like `a[i]=b[i]` in the body of a `for` loop in the complete range of both arrays can be removed together with one of those variables. This process allows removing a large number of equations and variables in a single step.

The command `./antialias` performs this task, removing aliases from a flat Modelica model and producing a flat alias–free Modelica model.

5.3 Reduction to μ-Modelica Syntax

The QSS Stand–Alone Solver accepts models described in a subset of the Modelica language called μ-Modelica, which has a reduced and restricted syntax.

At this stage, the Modelica statements that are not part of μ–Modelica are replaced by semantically equivalent expressions and operators.

The result of this stage is still an unsorted DAE system containing only μ–Modelica constructs.

The command `./mmo` performs this conversion, taking a flat Modelica model and producing an equivalent flat Modelica model with μ–Modelica supported syntax.

5.4 Causalization Stage

The causalization stage of ModelicaCC has two flavors:

- Classical causalization, where `for` loops are unrolled and array are expanded. This implementation is able to tackle algebraic loops using an external C function that solves them using the GSL library.
- Vectorized causalization, implementing the algorithm presented in Section 4 that attempts to preserve the arrays and `for` loops.

Both implementations use the Boost library for the implementation of the graph theory algorithms and the GiNaC library for symbolic equation manipulation.

The vectorized algorithm is used by default. In case it fails (due to an algebraic loop not yet handled), the classical algorithm is used.

The result of this stage is a causalized μ-Modelica model that can be translated into C and simulated by the QSS Solver.

The command `./causalize` performs this task, taking a flat unsorted μ–Modelica model and producing a sorted μ–Modelica system.

5.5 Simulation Code Generation

The QSS Solver is in charge of the last stage of the compilation pipeline, generating the C code from the μ–Modelica model, compiling it together with the numerical solver.

As it was already mentioned, the QSS Solver does not expand arrays or `for` loops producing a compact piece of C code that can be quickly handled by the C compiler (it uses the gcc, GNU Compiler Collection).

The QSS Solver allows to simulate the models with different numerical algorithms, including DASSL, DOPRI and all the QSS family.

6 Examples and Results

We report here the usage of the algorithms and tools developed on two large scale systems of varying size. In both cases, we are analyze the computational cost related to each stage of the compilation pipeline, comparing it with that of OpenModelica.

We also report simulation CPU time in order to assert the correctness and quality of the resulting C code.

Banchmark Platform We used OpenModelica 1.9.3+dev (r25437) and the QSS Solver in version rev.1299 on an Intel i7-3770 CPU @ 3.40GHz with a 64 bits Ubuntu Linux with 8Gb of RAM.

On both tools, we will use DASSL as numerical integration solver with a tolerance of $1e^{-6}$.

6.1 One Dimensional Heat Transfer

This model is a 1D Finite Difference discretization of a PDE heat transmission problem taken from (Sezginer, 2014-2015).

Table 1 shows the CPU time consumed by each stage of the pipeline by OpenModelica and ModelicaCC. There, the columns are **F**lattening, **S**orting, **C**ode Generation, **CO**de Compilation, **SIM**ulation.

Table 1. Timing (in sec.) of the compilation stages for different sizes of the Heat Transfer model

	OpenModelica			**ModelicaCC**			
Size	F+S+C	CO	SIM	F	S+C	CO	SIM
10	1.82	0.8	0.002	0.07	0.05	0.06	0.003
100	2.1	2.23	0.01	0.07	0.05	0.06	0.016
1K	7.9	8.18	0.5	0.07	0.05	0.06	0.8
4K	57.9	11.1	7.5	0.07	0.05	0.06	16.9
10K	316.9	26.7	–	0.07	0.05	0.06	–

We note that this model is flat already, so in the ModelicaCC case it could be directly fed to the causalization stage (since the whole pipeline is valid Modelica code) but we measure the trivial flattening stage anyway.

In the last row, with 10000 sections, DASSL fails to simulate.

6.2 A LC Transmission Line

The LC Line example is the one presented in Listing 1. Each section of the line is made with components from the MSL while the connections between these sections was done by writing the Modelica code.

Table 2 reports the CPU time of the different stages of the compilation.

Table 2. Timing (in sec.) of the compilation stages for different sizes of the LC Line model

	OpenModelica			**ModelicaCC**			
Size	F+S+C	CO	SIM	F	S+C	CO	SIM
10	0.1	1.0	0.006	0.03	0.04	0.2	0.005
100	1.9	1.4	0.14	0.03	0.04	0.2	0.094
1K	61.8	–	–	0.03	0.04	0.2	55
10K	–	–	–	0.03	0.04	0.2	–

Starting from 1000 components, the C compiler fails to compile the code generated by OpenModelica. For 10000 components, OpenModelica fails to generate the C code.

Again, for 10000 components DASSL fails to simulate the system.

6.3 Result Analysis

From the two examples studied above we see that the proposed algorithms and the ModelicaCC implementation have a constant complexity (both in space and in time) w.r.t. the model size. This not only allowed us to compile the models faster but we can handle larger models.

Notice that OpenModelica fails in the second case for 10000 components (which in fact correspond to more than 100000 variables, since the model contains several algebraic variable arrays). In that case, even for 1000 components, the C code produced is so large that it cannot be handled by the C compiler. All these problems disappear with the ModelicaCC approach.

In all cases, the simulation times are similar for both tools and the simulation results (not reported here) have a negligible difference. This tells us that DASSL is receiving equivalent set of equations. Moreover, when DASSL fails to simulate, DOPRI works fine with the QSS Solver.

In this benchmark we have only fully analyzed the open source OpenModelica tool. Anyway, we have also run the experiments on demo versions of Dymola and Wolfram System Modeler, obtaining a similar behavior when the model increases in size.

7 Conclusions and Future Research

In this article we presented novel algorithms for efficient compilation of large scale Modelica models. The idea behind them is to exploit the repetitive nature found on this kind of models and to process them without expanding iterative equations and array variables.

The two algorithms for Flattening and Causalization (Sec. 3 and 4) were implemented in a prototype C compiler called ModelicaCC. A design goal of ModelicaCC was to have a Modelica-valid pipeline throughout all the process and to exploit existing open source libraries.

The study performed on two large scale models shows that the algorithms and their implementation have a constant cost w.r.t. the model size while other Modelica tools have a supra-linear cost (both in flattening and code generation/compilation). This allows to compile arbitrary large models with no extra cost.

There are many directions for future work. First, the Vectorized Tarjan's algorithm must be extended to deal with algebraic loops. For higher index models, our idea is to extend Pantelides algorithm in the same way done for Tarjan's in order to reduce the index of non-expanded models.

As mentioned before, one possible source of large scale models is the spatial discretization of PDE's. The algorithms in this work are presented for unidimensional arrays and non nested loops. They must be extended to higher dimension and nested loops for their application in 2D and 3D discretizations.

Finally, so far we have made experiments with a few large scale models. All algorithms and tools presented here have to be tested on more complex cases.

The ModelicaCC compiler is an open source tool hosted at `www.sourceforge.net/projects/modelicacc/`.

The models used in this article can be downloaded from `www.fceia.unr.edu.ar/~fbergero/modelica15/`.

References

Matthias Arzt, Volker Waurich, and Jörg Wensch. Towards Utilizing Repeating Structures for Constant Time Compilation of Large Modelica Models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '14, pages 35–38, Berlin, Germany, 2014.

Dag Brück, Hilding Elmqvist, Sven Erik Mattsson, and Hans Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of Modelica 2002*, 2002.

Francesco Casella. Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives. In *11th International Modelica Conference*, 2015.

F. Cellier, X. F. Floros, and E. Kofman. The Complexity Crisis: Using Modeling and Simulation for System Level Analysis and Design. In *Proc. SimulTech 2013, 3rd International Conference on Simulation and Modeling Methodologies, Technologies, and Applications*, Reykjavik, Island, 2013.

F. E. Cellier and E. Kofman. *Continuous System Simulation.* Springer-Verlag, New York, 2006.

Joaquín Fernández and Ernesto Kofman. A Stand-alone Quantized State System Solver for Continuous System Simulation. *Simulation*, 90(7):782–799, July 2014. ISSN 0037-5497. doi:10.1177/0037549714536255. URL `http://dx.doi.org/10.1177/0037549714536255`.

Jens Frenkel, Christian Schubert, Gunter Kunze, Peter Fritzson, Martin Sjolund, and Adrian Pop. Towards a Benchmark Suite for Modelica Compilers: Large Models . In *8th Modelica Conference*, 2011.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-Interscience, New York, 2004.

Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, 2005.

John Hopcroft and Robert Tarjan. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM*, 16(6):372–378, June 1973. ISSN 0001-0782. doi:10.1145/362248.362272. URL `http://doi.acm.org/10.1145/362248.362272`.

Peter Fritzson Jens Frenkel, Gunter Kunze. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *9th Modelica Conference*, 2012.

Constantinos C. Pantelides. The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

L. R. Petzold. A description of DASSL: a differential/algebraic system solver. In *Scientific computing (Montreal, Quebec, 1982)*, pages 65–68. IMACS, New Brunswick, NJ, 1983.

Kaan Sezginer. A Test Suite of Large Scalable Models for Modelica Tool Evaluation. Master's thesis, POLITECNICO DI MILANO, 2014-2015.

Kristian Stavaker. *Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units.* PhD thesis, Linkopings Universitet, 2011.

Kristian Stavaker, Daniel Rolls, Jing Guo, Peter Fritzson, and Sven bodo Scholz. Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs. In *3rd EOOLT*, 2010.

Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1137/0201010.

Dirk Zimmer. Module-Preserving Compilation of Modelica Models . In *7th Modelica Conference*, 2009.