Set-Based Graph Methods for Fast Equation Sorting in Large DAE Systems

Pablo Zimmermann
FCEIA - Universidad Nacional de Rosario
Rosario, Argentina

Joaquín Fernández
CIFASIS - CONICET
Rosario, Argentina
fernandez@cifasis-conicet.gov.ar

Ernesto Kofman
CIFASIS - CONICET and FCEIA, UNR
Rosario, Argentina
kofman@fceia.unr.edu.ar

ABSTRACT

This paper introduces new algorithms for the efficient conversion of large sets of DAEs into ODEs based on the extension of maximum matching and Tarjan's strongly connected component algorithms using a novel concept of *Set–Based Graph*. These algorithms have the capability of solving the problems without expanding the arrays of unknowns and without unrolling the for-loop equations so that the complexity becomes independent on the size of the arrays. The implementation of the new algorithms in an experimental Modelica compiler is also described and two examples are presented.

CCS CONCEPTS

• **Mathematics of computing → Differential algebraic equations**; *Graph theory*; • **Computing methodologies → Modeling and simulation**; **Model development and analysis**.

KEYWORDS

Modelica, DAE sorting, large scale models, graph theory

ACM Reference Format:
Pablo Zimmermann, Joaquín Fernández, and Ernesto Kofman. 2019. Set-Based Graph Methods for Fast Equation Sorting in Large DAE Systems. In *Proceedings of EOOLT 2019.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The usage of modern object oriented mathematical modeling languages invariably leads to sets of differential algebraic equation systems. While there are DAE solvers that can directly deal with some of these models [9, 25], it is more efficient and robust to first transform the DAEs into ODEs and then use standard ODE solvers. This conversion, called causalization, is normally performed using graph–theory–based algorithms [9, 22].

Causalization algorithms decide which unknown has to be computed out of each equation or (matching) and they must also place the resulting causalized equations in a proper order inside the simulation code (sorting). In presence of algebraic loops, the algorithms should also find the minimal sets of unknowns that has to be simultaneously solved. For that goal, a bipartite graph is built where the nodes represent equations and unknowns while the edges represent the presence of an unknown in an equation. Then, the most used strategy is to perform first a matching procedure converting the bipartite graph into a directed graph, in which a topological sorting is obtained using Tarjan's strongly connected components algorithm [30]. This procedure has a linear computational complexity on the number of nodes and edges.

Models from diverse engineering domains contain arrays of equations, that can be the result of the spatial discretization of PDEs [5], or that can be due to the presence of large sets of simpler models, as it occurs in Buildings and Energy models (having many houses/rooms/consumers) [4, 15, 20], in particle dynamics [13, 27], crowd simulation [31] and ODE-based multi–agent models, amongst others. Moreover, these models can be also multidimensional (which is typical in 2D or 3D PDEs), and the number of individual equations and unknowns may become huge (of the order of several millions).

Therefore, even when the causalization procedure has a linear complexity with the number of equations and unknowns, its usage on these huge models becomes prohibitive and, in spite of some improvements proposed along the years [6, 16, 21], this fact remains as one of the main obstacles to simulate very large systems represented by sets of DAEs [8].

One particular modeling language that usually leads to problems of this type is Modelica [18, 23]. Modelica models are usually described by the coupling of simpler models described by sets of DAEs, but they can also contain large sets of equations inside loops (for-loop) relating arrays of unknowns.

The simulation of large scale Modelica models offers several difficulties, not only at the causalization, but also at the flattening and code generation stages [7]. A benchmark presented in [17] tested the efficiency of three Modelica simulation tools on large scale models showing that their processing time (at compilation phase) grows almost quadratically with the model size. A survey on different variations of the causalization algorithm is presented in [16], where the author analyses the different implementations on models with thousands of equations obtaining similar results to those of the previously cited work.

A crucial observation to overcome these problems is that very large models are rarely formed by thousands or millions of individual equations. They generally present some regularity that allows them to be iteratively described.

First attempts to exploit this feature at the index reduction phase were presented in [1] and recently extended to the causalization phase in [14], with both strategies being implemented in OpenModelica [19]. These solutions are based on collecting and collapsing repeated sub-structures of the incidence graph in order so that they are uniformly treated.

A different approach is followed in [3], where arrays of unknowns and for-loop equations are kept unexpanded at the flattening and causalization stages. The resulting flat and causalized Modelica model can be then used by the Stand-Alone QSS solver [12], that produces the C language simulation code without expanding arrays or unrolling for-loop equations. That way, this strategy achieves a computational cost throughout all the compilation pipeline that is independent on the size of the arrays involved. However, the work was limited to large scale models without algebraic loops and having one–dimensional arrays and some particular connection structures. A similar result for the causalization stage is presented in [29, Ch.9] and there is a tool called IDA Modelica that avoids the array expansion on certain sub-classes of Modelica models [26].

In this article, we formalize, extend and generalize the results of the work of [3] regarding the causalization stage. For that goal, we first introduce the concept of Set–Based Graphs (SB-Graphs) in which each vertex can represent an arbitrary set of equations or unknowns while the edges comprise the information about the appearance of sets of unknowns in sets of equations. Then, we extend maximum matching and Tarjan's algorithm for these types of graphs. In that way, in absence of structural singularities, the complete causalization stage can be completed without expanding arrays or unrolling for-loop equations.

We also describe the implementation of the algorithms as part of the experimental *Modelica C Compiler* (ModelicaCC) [3], that already incorporates a compact flattening procedure. The resulting compact causalized models can be then used by the stand-alone QSS solver, that can produce the simulation C language code without expanding arrays or unrolling for-loop equations. That way, the full Modelica compilation process (from flattening to C code generation) is performed with computational costs that are independent on the size of the arrays.

In order to demonstrate the advantages of the new approach, we show the usage of the algorithms in two different applications, where we compare the results obtained with the novel algorithm against those of standard expanding strategies.

2 PRELIMINARIES

2.1 Equation Sorting in Dynamical Systems

The transformation of a set of DAEs into a set of ODEs allows a more efficient usage of numerical integration algorithms. The problem can be stated as follows.

We consider a set of N equations $f_1, \dots, f_N$, and N unknowns $u_1, \dots, u_N$ of the form

$$\begin{aligned} f_1(u_1, \dots, u_N, t) &= 0 \\ &\vdots \\ f_N(u_1, \dots, u_N, t) &= 0 \end{aligned}$$

where some unknowns represent state derivatives, and where the functions f_i do not necessary depend on all the unknowns. The goal is to sort the set of equations both vertically and horizontally such that an execution order is obtained. This is, obtaining a BLT ordering of the form

$$\begin{aligned} u_{k(1)} &= g_1(t) \\ u_{k(2)} &= g_2(u_{k(1)}, t) \\ &\vdots \\ u_{k(N)} &= g_2(u_{k(1)}, \dots, u_{k(N-1)}, t) \end{aligned}$$

where some subsets of unknowns should be simultaneously solved (algebraic loops).

This problem is usually treated making use of graph theory procedures. For that goal, a bipartite graph is constructed associating a vertex to each unknown u_i and a vertex to each equation f_j. Then, the edges represent the appearance of each unknown on each equation. This is, an edge between the vertex f_j and the unknown u_i represents that unknown u_i appears in the expression of f_j.

After the bipartite graph is built, a maximum matching must be found. When the maximum matching is not found, the problem is possibly of higher index and an index reduction algorithm like Pantelides must be applied until the problem can be fully matched (or until discovering that it is structurally singular). The resulting maximum matching establishes which unknown is obtained from each equation.

Then, in order to establish the vertical sorting and the subsets of equations that must be simultaneously solved, a directed graph is built. Each vertex of the directed graph is obtained as the collapse of two matched vertices of the bipartite graph. Also, the unmatched edges of the bipartite graph are converted into directed edges of the directed graph. More precisely, assuming that f_i is matched with u_j and that f_k is matched with u_l, and that the expression of f_k also depends on u_j, then two collapsed vertices are created: fu_{ij} and fu_{kl} and a directed edge from fu_{kl} to fu_{ij} is included. This is, the unmatched edge connected f_k and u_j, then the direction of the resulting edge is from fu_{kl} to fu_{ij} (from equation to unknown). That way, the existence of a directed edge from f_{kl} to f_{ij} indicates that f_k cannot be solved before f_i.

Taking into account this last observation, the equations corresponding to each strongly connected component on the resulting directed graph must be simultaneously solved and the overall vertical sorting of the system of equations is given by the reverse topological sorting. The use of Tarjan's SCC algorithm solves both problems and it is then the usual tool for completing the causalization procedure.

Synthesizing, the whole procedure involves the construction of a bipartite graph, its maximum matching and collapsing and the use of Tarjan's algorithm on the resulting directed graph. We describe next these algorithms on which the novel results are based.

2.2 Maximum Matching Algorithms

In order to sketch the principles of maximum matching algorithms, we introduce first some definitions and notation:

- A graph is denoted as $G = (V, E)$ where $V = \{v_1, \dots, v_n\}$ is the set of vertices and $E = \{e_1, \dots, e_m\}$ is the set of edges. Each edge is a set of two vertices $e_i = \{v_k, v_l\}$.
- We consider in particular a bipartite graph in which $V = F \cup U$ where F is the set representing equations and U is the set representing unknowns, each of them having N vertices.

- In the algorithm below, we shall denote E_M to the set of matched edges.
- $V_M = \{v | \exists \tilde{v} : \{v, \tilde{v}\} \in E_M\}$ denotes the set of matched vertices.
- A *path* of length $n-1$ is a n–Tuple of vertices $p = (v^1, \ldots, v^n)$ with $n \geq 2$ such that for all $i \in [1, n-1]$, the set $\{v^i, v^{i+1}\} \in E$.
- An *alternating path* is a path $p = (v^1, \ldots, v^n)$ with the restriction that for all $i \in [2, n-1]$, $\{v^i, v^{i+1}\} \in E_M \iff \{v^{i-1}, v^i\} \notin E_M$.
- An *augmenting path* is an alternating path $p = (v^1, \ldots, v^n)$ where $v^1 \notin V_M$ and $v^n \notin V_M$.

A simple maximum matching algorithm can be then stated as follows:

```
$E_M \leftarrow \emptyset, V_M \leftarrow \emptyset.$
for $i = 1 : N$ do
    if $f_i \notin V_M$ then                                  ▷ unmatched edge
        Find an augmenting path $p = \{f_i, \ldots, u^j\}$
        $E_M \leftarrow E_M \oplus p$           ▷ Update set of matched edges[1]
        $V_M \leftarrow V_M \cup \{f_i, u_j\}$   ▷ Update set of matched vertices
    end if
end for
```

This algorithm obtains a maximum matching E_M finding augmenting paths from the unmatched vertices. Each augmenting path can be simply found performing a Depth First Search (DFS) following alternating paths until an unmatched vertex is found.

There are more efficient algorithms in the literature for solving this problem, but this simple procedure can be easily extended to set–based graphs as we shall see later in this work.

2.3 Strongly Connected Component Algorithms

After the matched bipartite graph is collapsed and converted into a directed graph, the vertical sorting requires finding the corresponding SCC and their reversal topological sorting. Both problems are simultaneously solved by Tarjan's algorithm sketched below, after providing the following definitions:

- A directed graph is denoted as $G_d = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ is the set of vertices and $E = \{e_1, \ldots, e_m\}$ is the set of directed edges. Each directed edge is a pair $e_i = (v_k, v_l)$.
- A strongly connected component of G_d is a maximal subset of vertices $\{v_1, \ldots, v_m\}$ where there is a path in each direction between each pair of vertices.

The next algorithm then obtains a tuple of SCC denoted V_s containing all vertices in G_d in reversal topological order. It makes use of a function that obtains a tuple of SCC starting from a single vertex.

```
$index \leftarrow 0$
$V_s \leftarrow \emptyset$                                  ▷ Initial tuple of SCC
$V_{ind} \leftarrow \emptyset$                               ▷ Set of indexed vertices
for $i = 1 : n$ do
    if $v_i \notin V_{ind}$ then                              ▷ Not indexed vertex
        $V_s \leftarrow V_s \circ SCC(v_i)$                    ▷ Add SCC from $v_i$
    end if
end for
```

```
function SCC($v$)
    $v.index \leftarrow index$
    $v.lowlink \leftarrow index$
    $index \leftarrow index + 1$
    $S.push(v)$
    $v.onstack \leftarrow true$
    $U \leftarrow \emptyset$                                 ▷ Local tuple of SCC
    for all $(v, w) \in E$ do
        if $w \notin V_{ind}$ then                     ▷ Not indexed successor
            $U \leftarrow U \circ SCC(w)$               ▷ Recurse on successor
            $v.lowlink \leftarrow \min(v.lowlink, w.lowlink)$
        else if $w.onstack$ then
            $v.lowlink \leftarrow \min(v.lowlink, w.index)$
        end if
    end for
    if $v.lowlink = v.index$ then              ▷ Root vertex of a SCC
        $S_C \leftarrow \emptyset$                            ▷ Create new SCC
        repeat
            $w = S.pop$
            $w.onstack = false$
            $S_C \leftarrow S_C \cup \{w\}$              ▷ Add $w$ to current SCC
        until $w = v$
        return $U \circ S_C$
    else
        return $\emptyset$
    end if
end function
```

[1]The symbol $\oplus$ denotes the symmetric difference set operation (everything that belongs to both sets individually, but does not belong to their intersection)

2.4 Sorted System of Equations

The final result V_s contains a tuple of SCCs in reverse topological sorting. This is, if a SCC S_i is located before another SSC S_j, then there are not edges from S_i to S_j meaning that the unknowns computed in S_j are not required by S_i and then the equations of S_i can be placed before in the final code.

2.5 The Modelica C Compiler

ModelicaCC [3] is a collection of open source tools used to translate a Modelica model into a subset of the Modelica language called μ-Modelica [12]. This sub-language is understood by the QSS Stand-Alone Solver [12], a simulation tool that has certain advantages to simulate large scale models: it does not unroll `for` loops when producing the final simulation code, it can use QSS integration methods that are convenient in some large scale cases [5], and it can execute simulations in parallel in a straightforward manner [11]. It contains different modules that perform the different compilation stages (flattening, anti–alias, causalization). Each module attempts to works without expanding arrays or unrolling `for` loops, so they can handle large scale models without suffering the problems of the other compilers.

Previous to the current work, a limitation of ModelicaCC was that it only worked with one–dimensional arrays and it could not handle nested `for` loops. Moreover, it did not handle general connection structures. This limitation was mainly due to the fact that the algorithms used in the causalization stage were not general [3]. In addition, it did not handle algebraic loops.

3 SET-BASED GRAPHS

The algorithms presented in this work are based on the use of *Set-Based Graphs* (SB-Graphs). This section introduces the concepts of undirected and directed graphs of this type.

3.1 Undirected Set-Based Graphs

The formalization of Set–Based Graphs is based on the following definitions:

Definition 1 (Set–Vertex). *A* Set–Vertex *is a set of vertices* $V = \{v_1, v_2, \ldots, v_n\}$.

Definition 2 (Set–Edge). *Given two Set–Vertices,* V^a *and* V^b*, with* $V^a \cap V^b = \emptyset$*, a Set–Edge connecting* V^a *and* V^b *is a set of non repeated edges* $E[\{V^a, V^b\}] = \{e_1, e_2, \ldots, e_n\}$ *where each edge is a set of two vertices* $e_i = \{v_k^a \in V^a, v_l^b \in V^b\}$.

Then, a *Set–Based Graph* is defined as follows:

Definition 3 (Set–Based Graph). *A Set–Based Graph is a pair* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where*

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ *is a set of disjoint set–vertices (i.e.,* $i \neq j \implies V^i \cap V^j = \emptyset$*).*
- $\mathcal{E} = \{E^1, \ldots, E^m\}$ *is a set of set–edges connecting set–vertices of* $\mathcal{V}$*, i.e.,* $E^i = E[\{V^a, V^b\}]$ *with* $V_a \in \mathcal{V}$ *and* $V_b \in \mathcal{V}$*. In addition, given two set edges* $E^i, E^j \in \mathcal{E}$ *with* $i \neq j$*, such that* $E^i = E[\{V^a, V^b\}]$ *and* $E^j = E[\{V^c, V^d\}]$*, then* $V^a \cup V^b \cup V^c \cup V^d \neq V^a \cup V^b$*. This is, two different set–edges in* $\mathcal{E}$ *cannot connect the same set–vertices.*

A particular case of Set–Based Graph is a bipartite Set–Based Graph defined as follows:

Definition 4 (Bipartite Set–Based Graph). *A Bipartite Set–Based Graph is a Set–Based Graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where two disjoints sets of set–vertices* $\mathcal{V}_1, \mathcal{V}_2$ *can be found verifying* $\mathcal{V}_1 \cup \mathcal{V}_2 = \mathcal{V}$*, such that for every edge* $E^i = E[\{V^a, V^b\}] \in \mathcal{E}$ *the condition* $V^a \in \mathcal{V}_i$ *implies that* $V^b \notin \mathcal{V}_i$

3.2 Directed Set-Based Graphs

Directed set-based graphs use directed set-edges, defined as follows:

Definition 5 (Directed Set–Edge). *Given two Set–Vertices,* V^a *and* V^b*, with* $V^a \cap V^b = \emptyset$ *or* $V^a = V^b$*, a directed Set–Edge from* V^a *to* V^b *is a set of non repeated edges* $E[(V^a, V^b)] = \{e_1, e_2, \ldots, e_n\}$ *where each edge is an ordered pair of vertices* $e_i = (v_k^a \in V^a, v_l^b \in V^b)$.

Then, a *Directed Set–Based Graph* is defined as follows:

Definition 6 (Directed Set–Based Graph). *A Directed Set–Based Graph is a pair* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *where*

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ *is a set of disjoint set–vertices (i.e.,* $i \neq j \implies V^i \cap V^j = \emptyset$*).*
- $\mathcal{E} = \{E^1, \ldots, E^m\}$ *is a set of directed set–edges connecting set–vertices of* $\mathcal{V}$*, i.e.,* $E^i = E[(V^a, V^b)]$ *with* $V_a \in \mathcal{V}$ *and* $V_b \in \mathcal{V}$*. In addition, given two set edges* $E^i, E^j \in \mathcal{E}$ *with* $i \neq j$*, such that* $E^i = E[(V^a, V^b)]$ *and* $E^j = E[(V^c, V^d)]$*, then either* $V^a \neq V^c$ *or* $V^b \neq V^d$*. This is, two different set–edges in* $\mathcal{E}$ *cannot connect the same set–vertices with the same direction.*

3.3 Relations between regular and set–based graphs

The following properties relating set–based graphs with regular graphs can be straightforwardly verified:

Proposition 1 (Regular Graph defined by a Set–Based Graph). *A (directed) set–based graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *with* $\mathcal{V} = \{V^1, \ldots, V^n\}$ *and* $\mathcal{E} = \{E^1, \ldots, E^m\}$ *defines a regular (directed) graph* $G = (V, E)$ *where*

$$V = \bigcup_{i=1}^{n} V^i; \quad E = \bigcup_{i=1}^{m} E^i$$

Proposition 2 (Bipartite Graph defined by a Bipartite Set–Based Graph). *A (directed) bipartite set–based graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ *defines a regular (directed) bipartite graph.*

Notice that Proposition 1 establishes that a set–based graph unambiguously defines a regular graph. However, a regular graph can be equivalent to several set–based graphs that contain the same elementary edges and vertices but grouped in a different way.

4 SET-BASED EQUATION SORTING

This section introduces the main results.

4.1 Bipartite Set-Based Graph Construction

We consider a set of DAEs described by a total of N scalar equations and N scalar unknowns, represented as follows:

- **Unknowns**: We assume that the unknowns are originally declared as different objects of class `Real`, class `Real[size]` or, in general, `Real[size_1, size_2, ..., size_m]` for multidimensional arrays. Then, we represent each object by an *unknown array* of the form $u^i = [u_1^i, \ldots, u_{s(i)}^i]$, where $s(i)$ is the size of the object u^i (i.e., the number of scalar components, with $s(i) = 1$ for scalar unknowns).
  Denoting with N_u the number of unknown arrays, the total number of scalar unknowns is then $N = \sum_{i=1}^{N_u} s(i)$.
- **Equations**: We assume that the model contains equations that can be inside (possibly nested) loops (for-loop). Each equation inside a for-loop is then represented by an *equation array* of the form $f^j = [f_1^j, \ldots, f_{r(j)}^j]$ where $r(j)$ is the size of the loop that f^j belongs to. Single equations (not belonging to loops) are represented in the same way, with $r(j) = 1$. Notice that all the components of each equation array share the same expression.
  Denoting with N_f the number of equation arrays, the total number of scalar equations can be computed as $N = \sum_{j=1}^{N_f} r(j)$.

Taking into account this DAE formulation, the corresponding bipartite SB–Graph is represented as follows:

- We associate a set–vertex $F^j = \{f_1^j, \ldots, f_{r(j)}^j\}$ to each equation array $f^j = [f_1^j, \ldots, f_{r(j)}^j]$.

- We associate a set–vertex $U^i = \{u^i_1, \ldots, u^i_{s(i)}\}$ to each unknown array $u^i = [u^i_1, \ldots, u^i_{s(i)}]$.
- For each scalar equation f^j_k and for each unknown u^i_l that appears in the expression of f^j_k, we associate a scalar edge $e_n = \{f^j_k, u^i_l\}$.
- Then, for each pair of set–vertices (F^i, U^j) such that there is at least one scalar edge connecting their components, we associate a set–edge containing the corresponding scalar edges $E^k = E(\{F^i, U^j\}) = \{e_n = \{f^j_k, u^i_l\} : f^j_k \in F^j, u^i_l \in U^i\}$.
- Finally, the resulting SB–Graph is $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \mathcal{F} \cup \mathcal{U}$, $\mathcal{F} = \{F^1, \ldots, F^{N_f}\}$, $\mathcal{U} = \{U^1, \ldots, U^{N_u}\}$, and $\mathcal{E} = \{E^1, \ldots, E^{N_e}\}$.

The following algorithm shows a procedure to build the bipartite SB–Graph that, under certain conditions, has a complexity that not depends on the size of the arrays $s(i)$ and $r(j)$.

(1) For each unknown array u^i create the corresponding set–vertex U^i.
(2) For each equation array f^j:
   - Create the corresponding set–vertex F^j.
   - For each unknown array u^i appearing in the expression of f^j, create the set edge $E(\{F^j, U^i\}) = \bigcup_{k=1}^{r(j)} \{f^j_k, u^i_{M^1_j(k)}\}$ where $M^1_j(k)$ is an index function representing that the k–th component of equation f^j involves the l–th component of array u^i, with $l = M^1_j(k)$.
   - If an unknown array u^i appears more than once in f^j (in expressions like `z[i]+z[i+1]+z[1]=...`) then add to the previous set–edge the sets $\bigcup_{k=1}^{r(j)} \{f^j_k, u^i_{M^q_j(k)}\}$ with the new functions $M^q_j(k)$, for $q = 2, \ldots$ according to different number of appearances of the unknown.

Notice that all set–vertices can be represented by intension. In addition, provided that we know an expression for functions $M^q_j(k)$ related the appearance of unknowns inside equations, the edges can be also represented by intension and the full graph can be built without taking into account the size of the arrays.

4.2 Maximum Matching Algorithm for Bipartite SB-Graphs

We introduce next a maximum matching algorithm for a bipartite SB–Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defined as in Sec.4.1. The algorithm uses the following definitions:

- $E_M \subseteq E = \bigcup_{k=1}^{N_e} E^k \in \mathcal{E}$ denotes the set of matched edges.
- A *set path* of length $n-1$ and size m is a set of disjoint paths $P = \{p_1, \ldots, p_m\}$ of the form $p_j = (v^1_j, \ldots, v^n_j)$ such that $\{v^i_1, \ldots, v^i_m\} \subseteq V^k \in \mathcal{V}$. This is, the i–th vertices of all paths belong to the same vertex–set.
- The i–th set of vertices of a set path P is denoted $V(P, i) = \{v^i_1, \ldots, v^i_m\}$.
- The i–th set–edge is denoted $E(P, i) = \{e^i_1, \ldots, e^i_m\}$ with $e^i_j = \{v^i_j, v^{i+1}_j\}$.
- An *alternating set path* is a set path where $E(P, i) \subseteq E_M \vee E(P, i) \cap E_M = \emptyset$, and, $E(P, i) \subseteq E_M \iff E(P, i+1) \cap E_M = \emptyset$.
- An *augmenting set path* is an alternating set path where $V(P, 1) \cap V_M = \emptyset$ and $V(P, n) \cap V_M = \emptyset$.

The matching algorithm can be then written as follows:

```
1:  $E_M \leftarrow \emptyset, V_M \leftarrow \emptyset$.
2:  for $i = 1 : N_f$ do
3:      $F \leftarrow \{f \in F^i | f \notin V_M\}$ ▷ Subset of unmatched equations of $F^i$
4:      while $F \neq \emptyset$ do
5:          Find a set augmenting path $P$ with $V(P, 1) = \tilde{F} \subseteq F$
6:          $E_M \leftarrow E_M \oplus \bigcup_{i=1}^{n} E(P, i)$    ▷ Update set of matched edges
7:          $V_M \leftarrow V_M \cup V(P, 1) \cup V(P, n)$ ▷ Update set of matched vertices
8:          $F \leftarrow F \setminus \tilde{F}$    ▷ Remove $\tilde{F}$ from the set of unmatched equations
9:      end while
10: end for
```

While the procedure looks simple, it has a tricky step in line 5 that defines the entire performance of the algorithm.

A trivial choice for $\tilde{F}$ would be taking a single unmatched equation $\tilde{F} = \{f\}$ for some $f \in F$. In that case, the entire procedure would result identical to that of a simple maximum matching algorithm for regular graphs. A simple way of doing this is to perform a DFS starting from the unmatched equation until an unmatched unknown through an alternating path is reached.

A more clever way of implementing the algorithm is to start from a large unmatched set of equations $\tilde{F} \subseteq F^i$ and then performing the DFS through alternating set paths, until an unmatched set of unknowns $\tilde{U}$ is found. In particular, one can start with the entire set–vertex $\tilde{F} = F$. The formalization of the procedure uses the following additional definitions:

- Given a set of vertices V^a and a set of edges E_c, we denote $V^b = R(V^a, E_c) = \{v | \exists \{v^a, v\} \in E_c \wedge v^a \in V^a\}$ to the reachable set from V^a through E_c.
- Given two sets of vertices V^a, V^b and a set of edges E_c with $R(V^a, E_c) \cap V^b \neq \emptyset$, we say that a set path P_d of length 1 is a set path from V^a to V^b through E_c provided that $V(P_d, 1) \subseteq V^a$, $V(P_d, 2) \subseteq V^b$, and $E(P_d, 1) \subseteq E_c$.
- Given a set of vertices $\tilde{V} \subseteq V$, we define the split operation $S(\tilde{V}, \mathcal{V}) = \{\tilde{V}^1, \ldots, \tilde{V}^m\}$ with $\bigcup_{i=1}^{n} \tilde{V}^i = V$ and $\tilde{V}^i \subseteq V^{k_i} \in \mathcal{V}$. This is, function S splits an arbitrary set of vertices into subsets of the set–vertices of the SB Graph.
- Given two paths $p_a = \{v^1_a, \ldots, v^n_a\}$, $p^b = \{v^1_b, \ldots, v^q_b\}$, with $v^n_a = v^1_b$, we denote the composed path $p_c = p_a \circ p_b = \{v^1_a, \ldots, v^n_a, v^2_b, \ldots, v^q_b\}$.
- Given two set paths, P^a and P^b with $V(P^a, n) \cap V(P^b, 1) \neq \emptyset$ where $n-1$ is the length of P^a, and where $V(P^a, i) \cap V(P^b, j) = \emptyset$ for $i \neq n$ or $j \neq 1$, we denote $P^c = P^a \circ P^b = \{p^c_1, \ldots, p^c_m\}$ to the set path of all possible composed path $p^c_i = p^a_j \circ p^b_k$.

Then, an augmenting set path can be found calling function $ASP(F, E \setminus E_M, \emptyset)$ defined below.

```
1: function $ASP(V, E_x, \hat{V})$
2:     $\hat{V} \leftarrow \hat{V} \cup V$    ▷ Set of visited vertices
3:     $\tilde{V} \leftarrow R(V, E_x) \setminus R(V, E_x) \cap \hat{V}$    ▷ Set of non–visited reachable vertices
4:     if $\tilde{V} = \emptyset$ then
5:         return $\emptyset$    ▷ Cannot reach unmatched unknowns
6:     else if $\tilde{V} \not\subseteq V_M$ then    ▷ Found unmatched unknonws
7:         Find a set path $P$ of length 1 from $V$ to $\tilde{V} \setminus (\tilde{V} \cap V_M)$ through $E_x$
8:         return $P$    ▷ Set path ending in unmatched unknowns
9:     else
10:        $(\tilde{V}^1, \ldots, \tilde{V}^m) \leftarrow S(\tilde{V}, \mathcal{V})$    ▷ Split $\tilde{V}$ into set vertex subsets
11:        for $k = 1, \ldots, m$ do
12:            $\tilde{P}_k \leftarrow ASP(\tilde{V}_k, E \setminus E_x, \hat{V})$    ▷ Search through alternating paths
13:            if $\tilde{P}_k \neq \emptyset$ then    ▷ Found unmatched unknowns
14:                Find a set path $\hat{P}$ of length 1 from $V$ to $V(\tilde{P}_k, 1)$ through $E_x$
15:                return $P = \hat{P} \circ \tilde{P}_k$    ▷ Set path ending in unmatched unknowns
16:            end if
17:        end for
18:        return $\emptyset$
19:    end if
20: end function
```

This algorithm returns an augmenting set path, unless the system is of higher index (or structurally singular). In such case, it returns $\emptyset$. It can be noticed that the operations involved do not depend on the size of the arrays.

4.3 SB-Graphs Collapsing

Once the matching process on the SB graph is completed, a SB directed graph is constructed collapsing the matched set–vertices and directing the non matched edges between collapsed vertices according to the type of vertices (equations or unknowns) they connected in the original bipartite graph. Each vertex of the directed graph is represented by a matched edge of the bipartite graph $fu^{ij} = \{f^i, u^j\} \in E_M$. Then, each directed edge is represented by an ordered pair of vertices, i.e., an ordered pair of edges of the bipartite graph $e_d^{ijkl} = (fu^{ij}, fu^{kl}) = (\{f^i, u^j\}, \{f^k, u^l\})$.

The algorithm for constructing a directed graph $\mathcal{G}_d(\mathcal{V}_d, \mathcal{E}_d)$ from a bipartite graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and a matching E_M is sketched below.

```
1: for all $E^k = E[F^i, U^j] \in \mathcal{E}$ do
2:     if $E_M \cap E^k \neq \emptyset$ then
3:         $FU^{ij} \leftarrow E_M \cap E^k$    ▷ The new vertex–set is the set of matched edges
4:         $\mathcal{V}_d \leftarrow \mathcal{V}_d \cup \{FU^i\}$
5:     end if
6: end for
7: for all $FU^{ij} \in \mathcal{V}_d$ do
8:     for all $FU^{kl} \in \mathcal{V}_d$ do
9:         $E_d^{ijkl} \leftarrow \{(\{f^i, u^j\}, \{f^k, u^l\})\}$ with $\{f^i, u^j\} \in FU^{ij}$, $\{f^k, u^l\} \in FU^{kl}$, and $\{f^i, u^l\} \in \bar{E}_M$    ▷ Directed set–edge
10:        if $E_d^{ijkl} \neq \emptyset$ then
11:            $\mathcal{E}_d \leftarrow \mathcal{E}_d \cup \{E_d^{ijkl}\}$
12:        end if
13:    end for
14: end for
```

4.4 SCC Algorithm for SB-Graphs

The problem of dividing the directed SB Graph into strongly connected components (SCC) is treated next. The procedure developed tries to finds sets of strongly connected components instead of finding them individually as Tarjan's algorithm does.

Before presenting the algorithm, we introduce some definitions. We consider a directed graph $\mathcal{G}_d(\mathcal{V}_d, \mathcal{E}_d)$ that defines a regular directed graph $G_d = (E_d, V_d)$ according to Prop.1. Then,

- A set-SCC of $\mathcal{G}_d$ is a set of Tuples of vertices $S_C = \{V^1, \ldots, V^n\}$ of the form $V^i = (v_1^i, \ldots, v_m^i)$ where $\bigcup_i v_j^i \subseteq V_d^k \in \mathcal{V}_d$ and where $\bigcup_j v_j^i$ is a SCC of G_d. This is, the vertices of the i–th Tuple of S_C belong to the same set–vertex and the vertices of the j–th component of each Tuple form a SCC in the equivalent regular graph.

With this definition, an SB-Graph extension of Tarjan's SCC algorithm that finds a tuple of set–SCCs $\mathcal{V}_s$ is sketched below

```
1: $index \leftarrow 0$
2: $\mathcal{V}_s \leftarrow \emptyset$    ▷ Initial tuple of set-SCC
3: $V_{ind} \leftarrow \emptyset$    ▷ Set of indexed vertices
4: for all $V \in \mathcal{V}_d$ do
5:     while $V \not\subseteq V_{ind}$ do ▷ There are non indexed vertices in $V$
6:         $\tilde{V} \leftarrow V \setminus V \cap V_{ind}$    ▷ Non indexed vertices in $V$
7:         $\tilde{\mathcal{V}}_s \leftarrow \emptyset$    ▷ Auxiliary tuple of set-SCC
8:         while $\tilde{\mathcal{V}}_s = \phi$ do
9:             $auxindex \leftarrow index$
10:            $\tilde{V}_{ind} \leftarrow V_{ind}$    ▷ Auxiliary set of indexed vertices
11:            $V_t \leftarrow$Tuple$(\tilde{V})$    ▷ Add order to $\tilde{V}$
12:            $S \leftarrow \emptyset$    ▷ Empty stack
13:            $(\tilde{\mathcal{V}}_s, \hat{V}) \leftarrow$ SSCC$(V_t)$    ▷ Find a tuple of set-SCC from $V_t$
14:            if $\tilde{\mathcal{V}}_s = \Phi$ then    ▷ Fail. $\tilde{V}$ was too large
15:                $\tilde{V} \leftarrow set(\hat{V})$  ▷ Try a smaller subset given by $\hat{V}$
16:                $index \leftarrow auxindex$
17:            else    ▷ Success
18:                $V_{ind} \leftarrow \tilde{V}_{ind}$    ▷ Update indexed set
19:                $\mathcal{V}_s \leftarrow \mathcal{V}_s \circ \tilde{\mathcal{V}}_s$    ▷ Add $\tilde{\mathcal{V}}_s$ to the tuple of set–SCC
20:            end if
21:        end while
22:    end while
23: end for
```

The algorithm above attempts to find tuples of set-SCC starting from each set-vertex. When it is not possible, it uses smaller subsets until all the vertices of the set–vertex are part of set-SCCs. The tuples of set-SCCs starting from a tuple of vertices V are found using a function SSCC() described below that, when fails, returns also a smaller tuple of vertices $\hat{V}$ that could be used for the next iteration.

```
1:  function SSCC($V$)
2:      $V.index \leftarrow index$
3:      $\tilde{V}_{ind} \leftarrow \tilde{V}_{ind} \cup V$                ▷ New indexed set
4:      $V.lowlink \leftarrow index$
5:      $index \leftarrow index + 1$
6:      $S.push(V)$
7:      $V.onstack \leftarrow true$
8:      $\mathcal{U} \leftarrow \emptyset$                          ▷ Local tuple of set-SCC
9:      $\{E_1, \ldots, E_n\} = splitedges(\mathcal{E}_d, V)$      ▷ split set edges into paths
10:     for $i = 1 : n$ do
11:         $W \leftarrow Succ(V, E_i)$ ▷ Immediate successors of $V$ through $E_i$
12:         $(W_{NI}, W_{ST}, W_{NS}) \leftarrow splitvertices(W)$     ▷ Split successors into non indexed vertices $W_{NI}$, vertices on stack $W_{ST}$ and remaining vertices (indexed but not on stack)
13:         if $W_{NI} \neq \emptyset$ then                  ▷ Not indexed set
14:             $(\tilde{\mathcal{U}}, \hat{W}) \leftarrow$ SSCC($W_{NI}$)
15:             if $\tilde{\mathcal{U}} = \Phi$ then
16:                 return $(\Phi, \hat{V} = pre(\hat{W}, , E_i))$     ▷ Fail, next time try $\hat{V} = pre(\hat{W}, E_i)$, such that $Succ(\hat{V}, E_i) = \hat{W}$.
17:             else
18:                 $\mathcal{U} \leftarrow \mathcal{U} \circ \tilde{\mathcal{U}}$ ▷ Add $\tilde{\mathcal{U}}$ to the tuple of set–SCCs
19:                 $V.lowlink \leftarrow \min(V.lowlink, W_{NI}.lowlink)$
20:             end if
21:         end if
22:         if $W_{ST} \neq \emptyset$ then             ▷ There are succesors in stack (possible cycle)
23:             Take $W_S \in S$ with $W_S \cap W_{ST} \neq \emptyset$   ▷ Tuple on stack
24:             if $W_{ST} \neq W_S$ then ▷ The tuple on stack is different.
25:                 $\hat{W} \leftarrow W_{ST} \cap W_S$        ▷ Common components of both tuples
26:                 return $(\Phi, \hat{V} = pre(\hat{W}, E_i))$      ▷ Fail, next time try $\hat{V}$.
27:             else
28:                 $V.lowlink \leftarrow \min(V.lowlink, W_{ST}.index)$       ▷ Cycle found
29:             end if
30:         end if
31:     end for
32:     if $V.lowlink = V.index$ then                   ▷ Root set vertex
33:         $S_C \leftarrow \emptyset$                           ▷ Create new set SCC
34:         repeat
35:             $W = S.pop$
36:             $W.onstack = false$
37:             $S_C \leftarrow S_C \cup \{W\}$          ▷ Add $W$ to current set SCC
38:         until $W = V$
39:         return $(\mathcal{U} \circ \{SC\}, V)$
40:     else
41:         return $(\emptyset, V)$                       ▷ Not a root set vertex
42:     end if
43: end function
```

The SSCC procedure above first splits the set–edges departing from V into set–edges defining paths of length 1 (this is, set of edges that connect non repeated vertices of two set vertices). Then, for each split set edge E_i, the algorithm splits the successors from V into the set of non indexed vertices (W_{NI}), the set of vertices on stack (W_{ST}) and the remaining vertices (that are indexed but not on stack, and are then to be ignored as in Tarjan's algorithm). After that, a DFS is performed through the non indexed successors.

Whenever a cycle is detected (i.e., there are indexed successors on stack) the algorithm checks that it involves the whole tuple in the right order. If it does not, it cannot establish for certain that it corresponds to a set of SCC and it returns $\emptyset$ indicating that the whole set cannot be treated. In addition, it returns a smaller tuple from which a set of cycles may be found.

When this algorithm is used with set-vertices containing single vertices and set–edges containing single edges, it is identical to Tarjan's. When using larger set–vertices, it can sometimes find sets of SCC. If it fails, it tries with smaller subsets until it reduces back to regular Tarjan's.

A particular property of the algorithm above is that each set–SCC detected cannot have edges between its regular SCC. The reason is that if an edge between two regular SCC exists inside a set SCC, the DFS will follow it and it will find a potential cycle that fails.

4.5 Sorted Equation Generation

The extended Tarjan algorithm produces a tuple of set SCC $\mathcal{V}_s$ with a reverse topological sorting. Recalling that a directed edge from a vertex a to vertex b indicates that the equations in vertex b must be computed first, the equations must be placed in the order they appear in $\mathcal{V}_s$ as in the regular case.

The only difference with the regular case is that each set–SCC goes inside a for-loop equation traversing the corresponding arrays. Taking into account that the set–SCCs found do not contain edges between their components, the order in which those arrays are traversed is irrelevant.

5 IMPLEMENTATION

This section briefly discusses some details regarding the implementation in ModelicaCC of the algorithms described in the previous section.

The implementation was completely developed in C++ based on different available open source libraries. In particular, the Boost Graph Library [28] is used to represent the graph structure, exploiting its capability to embed properties in vertices and edges to represent the corresponding F, U, and E sets. Also, the GiNaC library [2] is used to symbolically solve each model equation for the matched unknown. Other components of the Boost library are employed such as Spirit for parsing, Variant and Optional for the abstract syntax tree and the ICL for managing integer intervals.

5.1 SB-Graph Representation

Each set vertex is represented by a structure called Multi-Dimensional Interval (MDI). This structure is consists in a colection of hyperrectangles which has efficiently implemented several set operations including intersection, difference and boolean operations on set relations ($\subseteq$).

Set-edges are represented by a structure called Index Pair representing sets of one–to–one connections between components of two set–vertices. Each Index Pair is characterized by a function $M(i)$ that denotes the existence of a vertex of the form $\{f_i, u_{M(i)}\}$.

5.2 Maximum Matching Algorithm

The implementation of the maximum matching algorithm for SB-Graphs follows the algorithm described in Section 4.2. In addition, it implements an initialization heuristic that looks for vertices of degree one (i.e., that can only be matched with their immediate neighbors) and propose an initial matching for the entire vertex sets containing those vertices. This heuristics avoids that, in some cases, the algorithm unnecessary uses small sets in some stages.

5.3 SCC Algorithm

The implementation of the extended Tarjan's algorithm closely follows the procedure described in Section 4.4. In addition, when it finds a failure condition due to the existence of connections between the different cycles that may form a set–SCC, it first check if all those connections have the same direction and, it that case, it does not discard the cycles. That way, not only disconnected sets of SCC can be found and more general cases can be treated without falling back to smaller subsets.

5.4 Sorted Equation Generation

The code generation is almost identical to that of ModelicaCC using regular graphs. The only difference is that it adds for-loop headers around the sets of equations and the corresponding indexes to the unknowns involved inside them.

As in the regular ModelicaCC, the sets of equations associated to a SCC are first sent to GiNaC in order to solve them symbolically. If this fails, the code for solving the equations based on Newton iteration is automatically generated.

6 RESULTS

We introduce next two examples that illustrate the advantages of the proposed approach. There, we measured the execution time of the set-based and scalar causalization algorithms implemented in ModelicaCC. We could not isolate the causalization stage in OpenModelica in order to measure its performance. However, we did measure the total time until C code generation, which was very similar to that of scalar ModelicaCC plus the code generation of the Stand-Alone QSS solver, so we deduce that the causalization stage of OpenModelica takes a similar time to that of the scalar algorithm in ModelicaCC.

6.1 Example 1

The first example corresponds to the RLC network depicted in Figure 1.

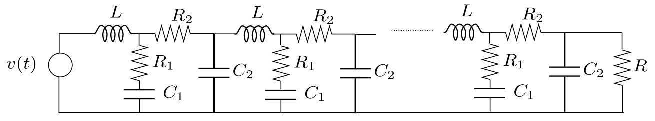

Figure 1: RLC Network

The flattened Modelica model with $N = 500$ is listed below. It can be easily checked that it produces a set of 500 independent algebraic loops between variables `IR1[i]`, `IR2[i]`, and `Ua[i]`.

```
 model rlc_loop
  constant Integer N = 500;
  parameter Real R1=1,R2=1,L=1,C1=1,C2=1,Vs=1,R=1;
  Real IR2[N], IL[N], UC1[N], Ua[N], IR1[N], UC2[N];
  Real VR, IR;
equation
  L*der(IL[1]) = Vs - Ua[1];
  C2*der(UC2[N]) = IR2[N] - IR;
  for i in 1:N loop
    IR1[i] = (Ua[i] - UC1[i])/R1;
    IR2[i] = (Ua[i] - UC2[i])/R2;
    IL[i] = IR1[i] + IR2[i];
    C1*der(UC1[i]) = IR1[i];
  end for;
  for i in 1:N-1 loop
    C2*der(UC2[i]) = IR2[i] - IL[i+1];
    L*der(IL[i+1]) = UC2[i]-Ua[i+1];
  end for;
  VR = R*IR;
  UC2[N] = VR;
end rlc_loop;
```

This set of equations is succesfully sorted by the set–based algorithms implemented in ModelicaCC. Table 1 reports the execution times of the novel algorithms against the standard ones as the number of RLC sections N is increased.

N	Scalar	Set–Based
10	0.05	0.16
100	0.17	0.15
1000	2.7	0.14
2000	6.29	0.15
5000	45.86	0.15
10000	96.27	0.15
20000	840	0.15
50000	6027	0.15
100000	16324	0.14
1000000	–	0.15

Table 1: Execution time (sec.) in the first example against model size for scalar and set–based algorithms

A simple analysis of these results shows that the scalar compilation of this model has an approximate complexity of $O(N^2)$. This is due to the fact that there are N algebraic loops that must be matched and sorted by Tarjan's algorithm. The maximum matching algorithm based on Edmonds-Karp [10], has $O(N^2)$ complexity explaining the poor overall performance. On the contrary, the complexity of the set–based algorithm is $O(1)$. In addition, the scalar approach produces a set of $6 \times N$ scalar equations that, for a large value of N, will eventually fail in the code generation stage or during the compilation of the resulting C++ code. Meanwhile, as we can see the code produced by the set–based algorithm does not depend on N:

```
for i in 1:1 loop
  IR1[i] = ((((R2*IL[i])+UC2[i])+(UC1[i]*(-1)))*(((R⌋
  ↪  2+R1)^(-1))));
end for;
for i in 1:1 loop
  IR2[i] = ((((UC2[i]+((R1*IL[i])*(-1)))+(UC1[i]*(-1⌋
  ↪  )))*(((R2+R1)^(-1))))*(-1));
end for;
for i in 1:1 loop
  Ua[i] = ((((R1*UC2[i])+((R2*R1)*IL[i]))+(R2*UC1[i]⌋
  ↪  ))*(((R2+R1)^(-1))));
  end for;
der(IL[1]) = (((L^(-1)))*(Vs+(Ua[1]*(-1))));
for i in 500:500 loop
Ua[i] = (((((R1*R2)*IL[i])+(UC1[i]*R2))+(R1*UC2[i]))⌋
↪  *(((R1+R2)^(-1))));
  end for;
for i in 500:500 loop
  IR1[i] = ((((R1+R2)^(-1)))*(((R2*IL[i])+(UC1[i]*(-⌋
  ↪  1)))+UC2[i]));
end for;
for i in 500:500 loop
  IR2[i] = (((UC1[i]+(R1*IL[i]))+(UC2[i]*(-1)))*(((R⌋
  ↪  1+R2)^(-1))));
end for;
VR = UC2[500];
IR = (((R^(-1)))*VR);
der(UC2[500]) =
↪  ((((C2^(-1)))*(IR+(IR2[500]*(-1))))*(-1));
for i in 2:499 loop
  IR2[i] = (((UC1[i]+(UC2[i]*(-1)))+(R1*IL[i]))*(((R⌋
  ↪  1+R2)^(-1))));
end for;
for i in 2:499 loop
  Ua[i] = (((((R1*IL[i])*R2)+(R1*UC2[i]))+(UC1[i]*R2⌋
  ↪  ))*(((R1+R2)^(-1))));
end for;
for i in 2:499 loop
  IR1[i] = ((((UC1[i]+((IL[i]*R2)*(-1)))+(UC2[i]*(-1⌋
  ↪  )))*(((R1+R2)^(-1))))*(-1));
end for;
for i in 1:500 loop
  der(UC1[i]) = (((C1^(-1)))*IR1[i]);
end for;
for i in 1:499 loop
  der(UC2[i]) =
  ↪  (((C2^(-1)))*(IR2[i]+(IL[i+1]*(-1))));
end for;
for i in 1:499 loop
  der(IL[i+1]) =
  ↪  ((((L^(-1)))*(Ua[i+1]+(UC2[i]*(-1))))*(-1));
end for;
```

6.2 Example 2

The second example is a model describing the temperature dynamics of an electrically heated rod, formulated as a 2D extension of the example described in Section 6.14 of [9]. The domain here is split into N radial sections by M longitudinal sections and it results in a system of $N \times M$ DAEs, with M independent nonlinear algebraic loops between the variables representing the border temperature and its time derivative.

For reasons of space, only a part of this model is listed below[2].

```
 model ELECTRICALLY_HEATED_ROD_MULTIDIM
  constant Integer N = 5, M = 5;
  Real T[N,M], T0[M], TL[M], dTdrR[M];
   ...
equation
    der(T[1,1])=omega*((T[2,1]-2*T[1,1]+T0[1])/delR+⌋
    ↪  (T[2,1]-T0[1])/(r*2*delR)+(T[1,2]-2*T[1,1]+T⌋
    ↪  room)/delX+p_elec/deltav);
  ....
  for i in 2:N-1,j in 2:M-1 loop
    der(T[i,j])=omega*((T[i+1,j]-2*T[i,j]+T[i-1,j])/⌋
    ↪  delR+(T[i+1,j]-T[i-1,j])/(r*2*delR)+(T[i,j+1⌋
    ↪  ]-2*T[i,j]+T[i,j-1])/delX+p_elec/deltav);
  end for;
  for j in 1:M loop
    T0[j]=4/3*T[1,j]-T[2,j];
    dTdrR[j]=-k1*(TL[j]^4-Troom^4)-k2*(TL[j]-Troom);
    dTdrR[j]=(-3*TL[j]-4*T[N,j]-T[N-1,j])/(2*delR);
  end for;
end ELECTRICALLY_HEATED_ROD_MULTIDIM;
```

This set of equations is successfully sorted by the set–based algorithms implemented in ModelicaCC. This time, GiNaC cannot find the analytical solution so the code for the Newton iteration is automatically produced. Table 2 reports the execution time of the causalization algorithms as the number of sections N, M is increased.

$N = M$	Scalar	Set–Based
10	0.109	0.148
100	8.46	0.157
200	33.7	0.167
500	206	0.171
1000	789	0.157
2000	–	0.174

Table 2: Execution time (sec.) in the second example against model size for scalar and set–based algorithms

A simple analysis of these results shows that the scalar compilation of this model has an approximate complexity of $O(N \times M + M^2)$. This is due to the fact that there are $N \times M$ equations and M algebraic loops that must be matched and sorted by Tarjan's algorithm. The scalar compilation does not work for values of $N \times M > 2 \times 10^6$. On the contrary, the complexity of the set–based algorithm is again

[2]the full model can be downloaded from https://github.com/CIFASIS/modelicacc/blob/experimental_multidim_intentional_causalization/test/ causalize/ELECTRICALLY_HEATED_ROD_MULTIDIM.mo

$O(1)$. As in the previous example, the code produced by the set–based algorithm does not depend on N or M.

7 CONCLUSIONS AND FUTURE RESEARCH

We introduced novel algorithms that extend maximum matching and Tarjan's SSC for the causalization of large sets of DAEs. These algorithms are based on a novel concept of *Set–Based Graphs* and allow to complete the causalization stage of Modelica models without expanding arrays or unrolling for-loop equations. That way, they achieve compilation times that are independent on the size of the arrays involved. The new algorithms were implemented in ModelicaCC and tested in different examples that showed the efficiency of the proposed solution.

This work opens several lines of research for the future. Besides the necessary generalizations and improvements (including better heuristics for some steps), we are currently working on the index–reduction problem by extending Pantelides algorithm [24] under similar principles. The flattening stage of the compilation process can also be extended in a similar way following what was done in [3]. The use of Set–Based Graphs can also be exploited for other related problems involving structure information on large DAE systems, such as Jacobian and incidence matrices computation.

Regarding the algorithms developed, it would be important to provide formal proofs of their correctness.

Also, a very important work for the future is to implement the algorithms in a more robust way in more robust Modelica compilers (OpenModelica in particular).

The source code of the set–based causalization can be downloaded from https://github.com/CIFASIS/modelicacc/tree/experimental_multidim_intentional_causalization.

8 ACKNOWLEDGMENTS

This work was partially funded by grants PICT–2017 2436 (ANPCYT) and PIP–2017/2019 11220170100485CO (CONICET).

REFERENCES

[1] Matthias Arzt, Volker Waurich, and Jörg Wensch. 2014. 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*. ACM, 35–38.

[2] Christian Bauer, Alexander Frink, and Richard Kreckel. 2002. Introduction to the GiNaC framework for symbolic computation within the C++ programming language. *Journal of Symbolic Computation* 33, 1 (2002), 1–12.

[3] Federico Bergero, Mariano Botta, Esteban Campostrini, and Ernesto Kofman. 2015. Efficient Compilation of Large Scale Modelica Models. In *11th International Modelica Conference - Versaille*.

[4] F. Bergero, F. Casella, E. Kofman, and J. Fernandez. 2018. On the Efficiency of Quantization–Based Integration Methods for Building Simulation. *Building Simulation* 11, 2 (2018), 405–418.

[5] Federico Bergero, Joaquín Fernández, Ernesto Kofman, and Margarita Portapila. 2016. Time discretization versus state quantization in the simulation of a one-dimensional advection-diffusion-reaction equation. *SIMULATION* 92, 1 (2016), 47–61.

[6] Emanuele Carpanzano and Claudio Maffezzoni. 1998. Symbolic manipulation techniques for model simplification in object-oriented modelling of large scale continuous systems. *Mathematics and Computers in Simulation* 48, 2 (1998), 133–150.

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

[8] François E Cellier, Xenofon Floros, and Ernesto Kofman. 2013. The Complexity Crisis.. In *ICSOFT*. IS–5.

[9] François E. Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer, New York.

[10] Jack Edmonds and Richard M Karp. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)* 19, 2 (1972), 248–264.

[11] J. Fernandez, F. Bergero, and E. Kofman. 2017. A Parallel Stand–Alone Quantized State System Solver for Continuous System Simulation. *J. Parallel and Distrib. Comput.* 106 (2017), 14–30.

[12] Joaquín Fernández and Ernesto Kofman. 2014. A Stand-alone Quantized State System Solver for Continuous System Simulation. *Simulation* 90, 7 (2014), 782–799.

[13] Amir Fijany, Tahir Çağin, Andres Jaramillo-Botero, and William Goddard III. 1998. Novel algorithms for massively parallel, long-term, simulation of molecular dynamics systems. *Advances in Engineering Software* 29, 3–6 (1998), 441 – 450.

[14] Martin Flehmig, Marcus Walther, Wolfgang E Nagel, Ines Gubsch, Joseph Schuchart, and Volker Waurich. 2015. Exploiting Repeated Structures and Vectorization in Modelica. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. Linköping University Electronic Press, 265–272.

[15] Xenofon Floros, Federico Bergero, Nicola Ceriani, Francesco Casella, Ernesto Kofman, and François Cellier. 2014. Simulation of smart-grid models using quantization-based integration methods. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. Linköping University Electronic Press, 787–797.

[16] Jens Frenkel, Günter Kunze, and Peter Fritzson. 2012. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*. Linköping University Electronic Press, 433–442.

[17] Jens Frenkel, Christian Schubert, Günter Kunze, Peter Fritzson, Martin Sjölund, and Adrian Pop. 2011. Towards a benchmark suite for Modelica compilers: Large models. In *Proceedings of the 8th International Modelica Conference; March 20th-22nd; Technical Univeristy; Dresden; Germany*. Linköping University Electronic Press, 143–152.

[18] Peter Fritzson. 2015. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: a Cyber-Physical Approach"*. Wiley-IEEE Press.

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

[20] Filip Jorissen, Michael Wetter, and Lieve Helsen. 2015. Simulation speed analysis and improvements of Modelica models for building energy simulation. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. Linköping University Electronic Press, 59–69.

[21] Fredrik Magnusson, Karl Berntorp, Björn Olofsson, and Johan Åkesson. 2014. Symbolic transformations of dynamic optimization problems. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. Linköping University Electronic Press, 1027–1036.

[22] Sven Erik Mattsson. 1995. Simulation of object-oriented continuous time models. *Mathematics and computers in simulation* 39, 5-6 (1995), 513–518.

[23] S. E. Mattsson, H. Elmqvist, and M. Otter. 1998. Physical system modeling with Modelica. *Control Engineering Practice* 6 (1998), 501–510.

[24] Constantinos C. Pantelides. 1988. The Consistent Initialization of Differential-Algebraic Systems. *SIAM J. Sci. Statist. Comput.* 9, 2 (1988), 213–231.

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

[26] Per Sahlin, Pavel Grozman, and Equa Simulation AB. 2003. IDA Simulation Environment-a tool for Modelica based enduser application deployment. In *Proceedings of the Third International Modelica Conference*. Citeseer.

[27] Lucio Santi, Nicolás Ponieman, Krzysztof Genser, Victor Daniel Elvira, Yung Jun Soon, and Rodrigo Castro. 2016. Application of state quantization-based methods in HEP particle transport simulation. In *22nd International Conference on Computing in High Energy and Nuclear Physics, CHEP 2016*. San Francisco, CA.

[28] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education.

[29] Kristian Stavåker. 2015. *Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures*. Ph.D. Dissertation. Linköping University Electronic Press.

[30] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[31] Armel Ulrich Kemloh Wagoum, Bernhard Steffen, Armin Seyfried, and Mohcine Chraibi. 2013. Parallel real time computation of large scale pedestrian evacuations. *Advances in Engineering Software* 60–61 (2013), 98 – 103. CIVIL-COMP: Parallel, Distributed, Grid and Cloud Computing.