Efficient Connection Processing in Equation–Based Object–Oriented Models.

Denise Marzorati[a], Joaquín Fernández[b*], Ernesto Kofman[a,b]

[a] *FCEIA-UNR, Argentina*
[b] *CIFASIS-CONICET, Argentina*

Abstract

This work introduces a novel methodology for transforming a large set of connections into the corresponding set of equations as required by the flattening stage of the compilation process of object oriented models. The proposed methodology uses a compact representation of the connections in the form of a *Set–Based Graph*, in which different sets of vertices and different sets of edges are formed exploiting the presence of regular structures. Using this compact representation, a novel algorithm is proposed to find the connected components of the *Set–Based Graph*. This algorithm, under certain restrictions, has the remarkable property of achieving constant computational costs with respect to the number of vertices and edges contained in each set. That way, under the mentioned restrictions, the proposed methodology can transform a large set of connections into the corresponding set of equations within a time that is independent on the size of the arrays contained in the model.

Besides describing the new algorithm and studying its computational cost, the work describes its implementation in a Modelica compiler and shows its application in different examples.

Keywords: Large Scale Models, Connected Components, Set–Based Graphs, Modelica

1. Introduction

Finding the connected components of an undirected graphs is a classic problem of Graph Theory that is employed in several application domains. Simple algorithms that solve this problem in linear time with the number of vertices have been known since several decades ago [12]. Also, parallel algorithms that can solve the problem in logarithmic time have been known for long time [11].

One particular problem that requires finding the connected components of a graph is found in the first stage of the compilation process that transforms

*Corresponding author
Email address: `fernandez@cifasis-conicet.gov.ar` (Joaquín Fernández[b])

an object oriented model [10] into executable simulation code. There, different sub-models are related by *connectors* that can contain two type of variables: those of *effort* type and those of *flow* type. Then, the connections must be replaced by equations expressing that all the connected variables of *effort* type are equal while the sum of the connected variables of *flow* type is equal to zero.

While finding the connected components in linear time may be affordable in several situations, there are models that are the result of the coupling of thousands of small sub-models where the cost can become prohibitive. Moreover, even if the problem is solved in a reasonable amount of time, the resulting system of equations can be so large that it is intractable by the subsequent stages of the compilation process.

Fortunately, large models often contain several repetitive connections that are the result of using `for` statements and this is a fact that can be exploited to reduce the computational cost of the different compilation stages [3, 16, 7, 4, 19, 5, 15, 1, 17]. However, the possibility of exploiting the presence of repetitive or regular structures at each stage requires that the previous stages had kept a compact representation. While there are some experimental implementations that in some particular cases can keep a compact representation during the whole compilation process [4], there is not yet a general solution.

Regarding the flattening stage, a general solution would require to find the sets of connected connectors which may be part of multidimensional arrays, solving the problem without actually expanding those arrays into individual connectors. This problem is equivalent to that of finding the connected components of an undirected graph while keeping some sets of vertices and edges grouped together, which constitutes the main goal of the present work.

The problem of manipulating large graphs grouping vertices and edges into sets to produce compact systems of equations was recently proposed with the introduction of *Set–Based Graphs* [20]. There, a compact solution for the problems of maximum matching and finding strongly connected components in directed graph for equation sorting was proposed and implemented as part of the prototype ModelicaCC compiler [4]. A more recent use of *Set–Based Graphs* was reported in [13], where the authors present a methodology that automatically obtains the compact code for computing the sparse Jacobian matrix of a differential algebraic equation model.

In this work, we use the same tool (Set-Based Graphs) and propose a general methodology for replacing connections by equations. For that purpose, we design an algorithm for finding connected components in undirected graphs. We show that, under certain assumptions, the computational cost of the algorithm becomes independent on the size of the sets of vertices and edges (i.e., the algorithm has a constant computational cost with the number of vertices and edges). In consequence, the cost of generating the set of equations in the flattening stage results independent on the size of the arrays of connectors.

Besides introducing and analyzing the methodology, we also describe its implementation in ModelicaCC. In addition, we analyze three examples (including a multidimensional one) showing the efficiency of the novel procedure and comparing its performance with that of OpenModelica.

The paper is organized as follows. After this introduction we briefly present a problem that motivates the work. Then, Section 2 introduces some concepts and previous works that are used as the basis of the proposed methodology and in Section 3 the novel Set-Based Graph algorithm for finding connected components is presented and the details of its prototype implementation in ModelicaCC are discussed. The new methodology for replacing connections by equations is then described in Section 4. Finally, Section 5 introduces some examples and Section 6 concludes the article.

1.1. Motivation

This work was motivated by a problem that appears in Modelica compilers. Modelica models can be represented by the coupling of several sub-models where the coupling is usually made using *connectors*. That way, the equations representing the structure of the circuit of Figure 1 can be represented by the piece of code in Listing 1.

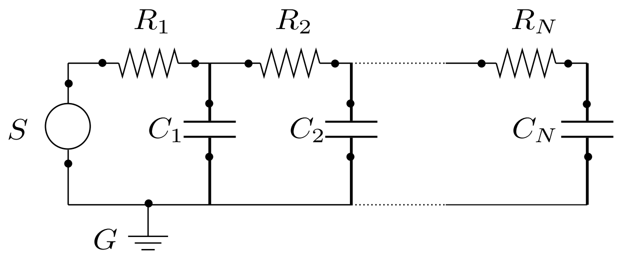


Figure 1: RC network

Listing 1: Modelica Connections

```
connect(S.p,R[1].p);
connect(S.n,G.p);
for i in 1:N-1 loop
  connect(R[i].n, R[i+1].p);
end for;
for i in 1:N loop
  connect(C[i].p, R[i].n);
  connect(C[i].n, G.p);
end for;
```

The connectors (`S.p`, `S.n`, etc) have two types of variables: *effort* variables that are equal to each other after being connected and *flow* variables whose sum is zero for all connected connectors. Thus, the resulting equations for the structure of Listing 1 would be that of Listing 2

Listing 2: Modelica Equations

```
S.p.effort=R[1].p.effort;
S.p.flow+R[1].p.flow=0;
S.n.effort=G.p.effort;
S.n.flow+G.p.flow+sum(C.n.flow)=0;
for i in 1:N-1 loop
  R[i].n.effort=R[i+1].p.effort;
  C[i].p.effort=R[i].n.effort;
  R[i].n.flow+R[i+1].p.flow+C[i].p.flow=0;
end for;
C[N].p.effort=R[N].n.effort;
R[N].n.flow+C[N].p.flow=0;
```

The translation from connections to equations requires finding connected components in a graph where the vertices represent the connectors (`S.p`, `S.n`, etc.) and the edges are defined by the presence of connections (`connect` statements) between the corresponding connectors.

Modelica compilers solve this problem by first expanding the `for` statements and the arrays of connectors and then finding the connected components and producing the equations as part of a process known as *flattening*. The result of this process in a model like that of Listing 1 is a large piece of code without the `for` statements of Listing 2. In addition, the cost of producing that code is at least linear with the size of the arrays involved (N in the above example).

When N is large (starting typically from 10^4 or 10^5) the computational costs become huge, and the length of the code produced may become intractable for the successive stages of the compilation process. Thus, we expect that the algorithms developed in this work provide a general solution for this problem as well as for other problems that require a compact and efficient connected components analysis in presence of some repetitive or regular structures.

2. Background

In this section we present some previous results and tools that are used along the rest of the paper.

2.1. Modelica and Equation-Based Object-Oriented Modeling Languages

In an effort to unify the different modeling languages used by the different modeling and simulation tools, a consortium of software companies and research groups proposed an open unified object oriented modeling language called *Modelica* [10, 8], that in the last two decades was progressively adopted by different modeling and simulation tools.

Modelica allows the representation of continuous time, discrete time, discrete event and hybrid systems. Elementary Modelica models are described by sets of differential and algebraic equations that can be combined with algorithms specifying discrete evolutions. These elementary models can be connected to other models to compose more complex models, facilitating the construction of multi–domain models.

Modelica models can be built and simulated using different software tools. OpenModelica [9] is the most complete open source package, while Dymola [6] and Wolfram System Modeler are the most used commercial tools. There are also some prototype tools oriented to different problems, such as JModelica [2] (for optimization problems) and ModelicaCC.

The simulation of Modelica models requires a previous compilation, that transforms the object oriented model description into a piece of code (usually in C language) containing a set of ordinary differential equations (ODE) or differential algebraic equations (DAE) that can be solved by an appropriate ODE or DAE solver. The compilation process is usually divided in several stages: flattening, alias removal, index reduction, equation sorting, and final code generation.

All Modelica compilers by default expand the arrays and unroll the `for loop` cycles in the first step of the compilation process. In consequence, in presence of large arrays, the computational cost of the compilation and the length of the produced code can become huge and the tools are unable to simulate systems with more that about 10^5 state variables. While there are some experimental implementations that avoid expanding and unrolling [4, 14], there is not yet a general solution.

2.2. Connected Components in Undirected Graphs

Finding the connected components of an undirected graph is a simple problem for which there are hundreds of algorithms. Linear time algorithms have been known since a long time ago [12], and there are also several parallel algorithms that can reduce the costs to logarithmic time. Among them, we shall briefly describe that of [11], which has certain features in common with the algorithm that constitutes the main result of this work.

This algorithm represents the connected components using a vector D of length n (the number of vertices in the graph) such that $D(i)$ contains the smallest numbered vertex in the connected component to which i belongs. A version of this procedure is described in Algorithm 1, where we consider that a graph $G = (V, E)$ is given with a set of vertices $V = \{1, 2, \dots, n\}$ and a set of edges $E = \{e_1, \dots, e_m\}$ with $e_k = \{i, j\}$ where $i, j \in V$.

Algorithm 1 Connected Components of [11]

```
 1: function CONNECT($V, E$)   ▷ All the steps are performed in parallel for all $i \in V$
 2:     $D(i) \leftarrow i$ for all $i \in V$.
 3:     for $it_1 = 1 : \log_2(n)$ do
 4:         $C(i) \leftarrow \min_j (D(j) | \{C(i), D(j)\} \in E \wedge D(j) \neq D(i))$, if none then $D(i)$, for all $i \in V$
 5:         $C(i) \leftarrow \min_j (C(j) | D(j) = i \wedge C(j) \neq i)$, if none then $D(i)$, for all $i \in V$
 6:         $D(i) \leftarrow C(i)$ for all $i \in V$.
 7:         for $it_2 = 1 : \log_2(n)$ do
 8:             $C(i) \leftarrow C(C(i))$ for all $i \in V$.
 9:         end for
10:         $D(i) \leftarrow \min(C(i), D(C(i)))$ for all $i \in V$.
11:     end for
12:     return $D$
13: end function
```

The details and the explanation of this algorithm is given in [11]. The algorithm we shall develop will use a very similar idea to represent the connected components (with a more general idea of the vertex numbering) and we shall also use an auxiliary vector like $C(i)$ with a similar idea for merging the components in step 4 and applying the map into itself like in step 8 until all the members of a component point to the same root vertex.

2.3. Set–Based Graphs

The algorithms presented in this work are based on the use of *Set-Based Graphs* (SB-Graphs), first defined in [20]. SB-Graphs are regular graphs in which the vertices and edges are grouped in sets allowing sometimes a compact representation. We introduced next the main definitions.

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

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 $E[\{V^a, V^b\}] = \{e_1, e_2, \dots, e_n\}$ where each edge is a set of two vertices $e_i = \{v_k^a \in V^a, v_l^b \in V^b\}$.*

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

- *$\mathcal{V} = \{V^1, \dots, V^n\}$ is a set of disjoint set–vertices (i.e., $i \neq j \implies V^i \cap V^j = \emptyset$).*
- *$\mathcal{E} = \{E^1, \dots, E^m\}$ 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.*

As in regular graphs, we can define bipartite Set–Based Graph and directed Set–Based Graphs. An algorithm for matching in bipartite Set–Based Graph and an algorithm for finding the strongly connected components of a directed Set–based Graph were recently presented in [20].

An SB-Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defines an equivalent regular graph $G = (V, E)$ where $V = \bigcup V^i \in \mathcal{V}$ and $E = \bigcup E^i \in \mathcal{E}$. Thus, a SB–Graph contains the same information than a regular graph. However, SB-Graphs can have a compact representation of that information provided that every set–edge and every set-vertex is defined by *intension.*

3. Connected Components in Set–Based Graphs

This section introduces a novel algorithm that finds the connected components of a Set–Based Graph. We first introduce a simple but inefficient algorithm for finding the connected components of ordinary graphs. Then we show that this algorithm, in the context of Set–Based Graphs, can be implemented using compact operations on some sets and maps leading to computational costs that, under certain circumstances, become independent on the number of vertices and edges.

3.1. An Inefficient Algorithm for Regular Graphs

We present first an algorithm for computing the connected components in a ordinary graph $G = (V, E)$. The proposed algorithm finds a collection of connected components represented in a similar way to that Algorithm 1. In particular:

- We assume that there exists a total ordering between all individual vertices (they could be represented by integer numbers, by arrays of integer numbers, etc).
- Each connected component is represented by one of its vertices $v_k \in V$, which is the smallest vertex of the connected component.
- There is a map $D_{\text{map}} : V \to V$ such that $D_{\text{map}}(v_r) = v_k$ implies that the vertex $v_r \in V$ is part of the connected component represented by v_k.
- Since the representative $D_{\text{map}}(v_r)$ is the minimum vertex on the connected component, then $D_{\text{map}}(v_r) \leq v_r$ for all $v_r \in V$.

Making use of this representation, Algorithm 2 finds the connected components represented by D_{map} of an arbitrary graph $G = (V, E)$.

The algorithm works as follows. It starts assuming that all vertices are disconnected so they represent their own connected component. Then, it iterates until the image of D_{map} becomes constant, meaning that no further components can be connected.

During each iteration a new map C_{map} is computed by adding connections between components. For each component represented by v_r, the algorithm takes into account all the edges connecting vertices of this component. Among all these edges, it takes the one that connects to certain vertex v_b with the least representative $v_k = D_{\text{map}}(v_b)$ (it could happen that $v_k = v_r$ if there is no connection from the component represented by v_r to any component represented by a smaller vertex). Then, if the representative v_k is smaller than v_r, the algorithm connects both components by making $C_{\text{map}}(v_r) = v_k$. In that case, it also reconnects all the vertices that were connected to v_r such that they are now connected to v_k.

Although it could be easily proved that the procedure is correct, it is possibly one of the less efficient algorithms one can imagine to find connected components in a graph. Its computational cost appears to grow at least quadratically with the number of vertices and edges. However, we shall see next that in the context of Set–Based Graph this algorithm can be implemented in a way that the costs become independent on the size of the different sets involved.

A key feature of the algorithm above that will allow this simplification is that in each iteration C_{map} is computed as a function of the complete map D_{map} and vice-versa. That way, both maps can be entirely computed from each other in simple steps.

Algorithm 2 Connected Components

```
 1: function CONNECT($V, E$)
 2:     $D_{\text{map}} \leftarrow \text{Identity}_{\text{map}} : V \to V$    ▷ All vertices are initially disconnected
 3:     $I_{\text{old}} \leftarrow \emptyset$    ▷ Previous image set of $D_{\text{map}}$
 4:     while $I_{\text{old}} \neq \text{Image}(D_{\text{map}})$ do
 5:         $C_{\text{map}} \leftarrow D_{\text{map}}$    ▷ New map of connected components
 6:         for all $v_r \in \text{Image}(D_{\text{map}})$ do    ▷ Component represented by $v_r$
 7:             if $\exists\{v_r, v_s\} \in E$ then    ▷ $v_r$ is not an isolated vertex
 8:                 $v_k \leftarrow \min(D_{\text{map}}(v_b) : (\{v_a, v_b\} \in E \wedge D_{\text{map}}(v_a) = v_r))$    ▷ Minimum component connected to the component represented by $v_r$
 9:                 if $v_k < v_r$ then
10:                     $C_{\text{map}}(v_r) \leftarrow v_k$    ▷ Connect components represented by $v_r$ and $v_k$
11:                     $C_{\text{map}}(v_a) \leftarrow C_{\text{map}} \circ C_{\text{map}}(v_a) = C_{\text{map}}(v_r) = v_k$ for all $v_a : C_{\text{map}}(v_a) = v_r$    ▷ All components represented by $v_r$ are now represented by $v_k$
12:                 end if
13:             end if
14:         end for
15:         $I_{\text{old}} \leftarrow \text{Image}(D_{\text{map}})$    ▷ Image of the previously connected components
16:         $D_{\text{map}} \leftarrow C_{\text{map}}$    ▷ New map of connected components
17:     end while
18:     return $D_{\text{map}}$
19: end function
```

3.2. Set–Based Graph Algorithm

The goal of using Set–Based Graph is to exploit the presence of repeating regular structures along the graph, representing the different sets by intension. While the definitions of SB–Graphs do not explicitly establish this, we propose next a simple way of representing the set edges that allows the intensive treatment of the graph.

Let E^h be a set-edge connecting V^i and V^j. We shall characterize this set–edge using two maps that relate the individual edges $e^h_k \in E$ with the vertices it connects $v^i_r = \text{map}^{h,i}(e^h_k)$ and $v^j_s = \text{map}^{h,j}(e^h_k)$. This is, the set edge is compactly defined as

$$E^h = \bigcup_k \{v^i_r = \text{map}^{h,i}(e^h_k), v^j_s = \text{map}^{h,j}(e^h_k)\}.$$

Thus, provided that there is a compact expression for these maps and that the set-vertices are represented by intension, the complete SB–Graph has a compact representation.

Using this representation of an SB–Graph, the previous algorithm can be reformulated as proposed in Algorithm 3.

Algorithm 3 Connected Components with SB–Graphs

```
1: function ConnectSBG($\mathcal{V}, \mathcal{E}$)
2:     $V \leftarrow \bigcup V^i \in \mathcal{V}$    ▷ Set of all vertices
3:     $(E^1_{\text{map}}, E^2_{\text{map}}) \leftarrow \text{edgeMaps}(\mathcal{E})$    ▷ First and second maps from edges to vertices
4:     $D_{\text{map}} \leftarrow \text{Identity}_{\text{map}} : V \to V$    ▷ All vertices are initially disconnected
5:     $I_{\text{old}} \leftarrow \emptyset$    ▷ Previous image set of $D_{\text{map}}$
6:     while $I_{\text{old}} \neq \text{Image}(D_{\text{map}})$ do
7:         $ER^1_{\text{map}} \leftarrow D_{\text{map}} \circ E^1_{\text{map}}$    ▷ First map from edges to connected components
8:         $ER^2_{\text{map}} \leftarrow D_{\text{map}} \circ E^2_{\text{map}}$    ▷ Second map from edges to connected components
9:         $C^1_{\text{map}} \leftarrow \text{minAdjMap}(ER^1_{\text{map}}, ER^2_{\text{map}})$    ▷ Map from components to least components via $E^2_{\text{map}}$
10:        $C^2_{\text{map}} \leftarrow \text{minAdjMap}(ER^2_{\text{map}}, ER^1_{\text{map}})$    ▷ Map from components to least components via $E^1_{\text{map}}$
11:        $C_{\text{map}} \leftarrow \min(D_{\text{map}}, C^1_{\text{map}}, C^2_{\text{map}})$    ▷ Map from components to least components
12:        $I_{\text{old}} \leftarrow \text{Image}(D_{\text{map}})$    ▷ Image of the previously connected components
13:        $D_{\text{map}} \leftarrow (C_{\text{map}})^\infty$    ▷ New map of connected components
14:    end while
15:    return $D_{\text{map}}$
16: end function
```

In this new algorithm, we made use of the following functions and notation:

- Function edgeMaps($\mathcal{E}$) returns two maps: a map of first connections $E^1_{\text{map}} : E \to V$ and a map of second connections $E^2_{\text{map}} : E \to V$, defined as follows. For each set–edge $E^h \in \mathcal{E}$ connecting set vertices V^i, V^j, the maps $E^{1,2}_{\text{map}}$ satisfy

$$E^1_{\text{map}}(e^h_k) = map^{h,i}(e^h_k) \forall e^h_k \in E^h$$
$$E^2_{\text{map}}(e^h_k) = map^{h,j}(e^h_k) \forall e^h_k \in E^h$$

  Notice that for each set edge, there are two possible definitions of E^1_{map} and E^2_{map}, according to which one is associated with i and which one with j (the set–edges are non–directed).

- Function minAdjMap($\text{map}_1, \text{map}_2$) computes a map map_3 such that

$$\text{map}_3(v) = \min(\text{map}_2(e) : \text{map}_1(e) = v) \quad (1)$$

  In the context of this algorithm, v is a representative vertex and e is an edge. Thus, for all edges such that $\text{map}_1(e) = v$, the function takes the one for which $\text{map}_2(e)$ is minimum and defines $\text{map}_3(v) = \text{map}_2(e)$. That way, $\text{map}_3(v)$ is the least representative vertex connected via map_2 to a vertex represented by v.

In the algorithm, the function is invoked twice with the inverted arguments in order to find the least representative connected to a component via both maps.

- The notation $(C_{\text{map}})^\infty$ is the result of applying C_{map} on itself until arriving to a fixed point.

The algorithm is almost identical to the previous one, except that the iteration of C_{map} on itself (step 11 in Algorithm 2) is now performed at the end of the cycle. The convergence of this new iteration is ensured by the fact that C_{map} is always less or equal than the identity map and that its domain is finite (V).

3.3. About the Computational Costs

We shall see in the next section that, under certain assumptions on the definition of the maps, all the steps involved in this new algorithm can be computed by intension (including the infinite iteration of C_{map} on itself). Then, the computational cost of each iteration of the algorithm (steps 6–14) becomes independent on the size of the sets.

Regarding the number of iterations that are actually needed until all components are connected, the following result establishes an upper bound.

Lemma 1. *The numbers of iterations required to find all connected components is at most* $2\log_2(N)$*, where* N *is the number of vertices in the largest connected component.*

Proof. Suppose that after certain number of iterations k, a component represented by v_r contains one or more connections to other components represented by v_{s_1}, v_{s_2}, etc. Suppose also that during the next iteration the component represented by v_r is not connected to any of those components.

If that occurs is because $v_r < v_{s_i}$ (otherwise it would be connected to the component represented by the minimum v_{s_i}). In addition, the components represented by v_{s_i} will be connected in that iteration to some components represented by $v_{t_j} < v_r$ (otherwise, they would be connected to the component represented by v_r). Then, in the following iteration, v_r will have connections to components represented by $v_{t_j} < v_r$ and it will be connected to the least v_{t_j}.

Thus, every component containing connections to other components is always connected after a maximum of two iterations. It means that after two iterations the number of different components that will be part of the same connected component is reduced at least to the half and they will be reduced to a single component after at most $2\log_2(N)$ iterations. □

This lemma tells that the number of iterations (and so the computational costs) of the algorithm may actually depend on the size of the sets. However, in several cases it does not:

1. When the structure is such that each connected component can only have a bounded number of vertices (independently of the size of the set-vertices).

2. When the latter condition is not accomplished by some connected components, but each connected component can be split in two components: the first one verifying the previous condition and the second one having all its vertices disconnected among them but connected to some vertices of the first component.

3. When the second component of the previous case has edges among its vertices forming an ordered path $(v_{r_1} - v_{r_2} - v_{r_3} - \ldots - v_{r_p})$ with $v_{r_1} < v_{r_2} < v_{r_3} < \ldots < v_{r_p}$.

The independence of the computational costs with the size of the sets in the first case is ensured by Lemma 1.

In the second case, the fact that the *large* set of edges has only connections to the small set of edges implies that in at most two iterations the edges of the large set will be connected to the edges of the small set (the reason for that can be found in the proof of Lemma 1). After that, the number of components is reduced to a quantity that is independent on the size of the sets and so is the number of additional iterations.

In the third case, a set of connections of the form $v_{r_1} - v_{r_2} - v_{r_3} - \ldots - v_{r_p}$ with $v_{r_1} < v_{r_2} < v_{r_3} < \ldots < v_{r_p}$ produces that all the components get connected in a single iteration of the algorithm (unless they are first connected to the small set of components). Then, in either situation, the case reduces to the situation analyzed in the previous case.

In conclusion, the only situation in which a large number of iterations would be required is under the presence of a large connected component resulting from a large non–ordered set of connections. Yet, that would be only possible when the maps that define the set edges have some irregular definition.

3.4. Algorithm Implementation

Algorithm 3 was implemented as part of the ModelicaCC [1] compiler, a tool that provides an environment to developed and tested novel algorithms involved in the different compilation stages of large scale Modelica models. For this implementation, we developed a `C++ SBGraph` library that contains all the required data structures, including the concepts of `Interval`, `Set`, `Map`, and `SBGraph`, and the different operations involving them. We describe next their main features.

3.4.1. Intervals

A uni-dimensional interval is represented by three natural numbers: `Interval.start`, `Interval.step`, and `Interval.end`. For instance, the se-

[1] Implementation source code can be found at: `https://github.com/CIFASIS/modelicacc/releases/tag/v3.0`

quence $[3, 5, 7, \ldots, 199]$ is compactly represented by `start`=3, `step`=2, and `end`=199 (we shall simply denote it by $[3 : 2 : 199]$).

A general interval of dimension d (denoted as `MultiInterval`) is represented by a list containing d intervals. For instance, the sequence

$$[(1; 1), (1; 2), \ldots, (1; 100), (4; 1), (4; 2), \ldots, (4; 100), \ldots, (1000; 1), (1000; 2), \ldots, (1000; 100)]$$

is represented by a a list containing two intervals described as: `Interval.start` = 1, `Interval.step` = 1, `Interval.end` = 100 and `Interval.start` = 1, `Interval.step` = 3, `Interval.end` = 1000. We shall denote it by $[1 : 1 : 100] \times [1 : 3 : 1000]$.

On these intervals we defined some basic functions and operations used by the higher level class that defines sets.

3.4.2. Sets

A set is defined as list of *atomic sets* represented by intervals of the same dimension. This is, `Set.AtomSets`(1) contains the first interval, `Set.AtomSets`(2) contains the second interval, etc. For instance, the set

$$S = \{2, 4, 6, \ldots, 100\} \cup \{101, 102, \ldots, 200\}$$

is represented by the union of two atomic sets represented by the intervals: $[2 : 2 : 100]$ and $[101 : 1 : 200]$ and we shall denote it as $S = \{[2 : 2 : 100]\} \cup \{[101 : 1 : 200]\}$.

On the set class, we defined some functions and operators, including the basic operations `setUnion`, `setIntersection`, and `setMinus`. All the operations are computed by *intension* using only the `start`, `step` and `end` values of the underlying intervals, and the result is another set represented by intervals. That way, the cost of the operations is independent on the size of the intervals involved.

Notice that the set representation is not canonical, so the same set can have alternative representations as the union of different atomic sets. Thus, in order to check is two sets A and B are equal, we actually check if $A \setminus B = B \setminus A = \emptyset$.

3.4.3. Maps

A one dimensional linear map is defined by two rational numbers: `linearMap.gain` (which cannot be negative) and `linearMap.offset`. Similarly, a general d–dimensional linear map is defined by two lists of length d `linearMap.gain`$(1 : d)$, and `linearMap.offset`$(1 : d)$.

A `Map` is then defined by a list of disjoint sets `Map.domain`$(1 : M)$ and a list of linear maps `Map.linearMap`$(1 : M)$, where all the sets and linear maps have the same dimension. For instance, a map like

$$i = \begin{cases} j + 3 & \text{for } j \in \{1, 2, \ldots, 100\} \\ 100 & \text{for } j \in \{101, 103, \ldots, 199\} \\ j/2 & \text{for } j \in \{102, 104, \ldots, 200\} \end{cases}$$

is defined by

- `Map.domain(1)`=$\{1 : 1 : 100\}$, `Map.linearMap(1).gain`=1, `Map.linearMap(1).offset`=3
- `Map.domain(2)`=$\{101 : 2 : 199\}$, `Map.linearMap(2).gain`=0, `Map.linearMap(2).offset`=100
- `Map.domain(3)`=$\{102 : 2 : 200\}$, `Map.linearMap(1).gain`=1/2, `Map.linearMap(1).offset`=0

A restriction in the definition of a map is that every domain and its correspondent linear map must be such that the resulting image in each dimension is composed by natural numbers. Thus, when a gain is not an integer number, the corresponding domain and offset cannot be arbitrary. Otherwise, if the gain is integer, the offset must be integer too.

On these maps we also implemented several functions and operators. Among them, we mention the following ones:

- `imageMap` computes the set that is the image of a given set through a given map. Similarly, `preImageMap` computes the preimage set.
- `compMaps` computes the new map that results from composing two maps ($\mathrm{map}_3 = \mathrm{map}_1 \circ \mathrm{map}_2$).
- `minMap` computes the minimum map between two maps, i.e., $\mathrm{map}_3(v) = \min(\mathrm{map}_1(v), \mathrm{map}_2(v))$, which can result equal to map_1 in some subdomain, and equal to map_2 in the remaining subdomain.

  This function requires establishing an ordering between the elements. For one dimensional sets the ordering is that of the natural numbers. For higher dimensional sets, the order between two elements is established at the first dimension in which they differ. This is, we say that $v < w$ if $v_1 < w_1$ or $v_1 = w_1 \wedge v_2 < w_2$, etc.
- `minAdjMap`: Given two maps map_1 and map_2 with the same domain, this function computes a new map map_3 according to Eq.(1). The computation of the new function is based on the following observation:
  - If map_1 is bijective, then map_3 can be computed as $\mathrm{map}_2 \circ \mathrm{map}_1^{-1}$.
  - If map_1 is constant, then map_3 can be computed as $\mathrm{map}_3(v) = \min(\mathrm{map}_2(e))$ for all e in the domain of the maps.

  Then, the function is implemented computing on each sub-domain and on each dimension of map_1 according to the previous observation.
- `mapInf`: Consider a map map_1 with the following restrictions:
  - All its linear maps have gains (in all the dimensions) that can only take the values 1 and 0.
  - If a gain is 1, the corresponding offset cannot be positive.

On this map, this function computes a new map map_2 that is the result of composing map_1 with itself until reaching convergence. The computations are performed without actually iterating on map_1. Instead, it computes the fixed points of the iteration and the maps to those fixed points.

The implementation is based on the following observations:

– A domain where the map has gain 1 and offset 0 remains unchanged after each iteration.

– If all domains have gain 0, then the iteration converges after at most N steps where N is the number of domains.

– If a domain has gain 1 and offset -1, then after some iterations of the map it will take a value outside the domain (`interval.start` -1 in fact). Thus we can just replace the gain by 0 and the offset pointing to `interval.start` -1.

– If a domain has gain 1 and offset -2, we shall have two arrival points after leaving the domain. So we can split the interval in two intervals with gain 0 and different offset. For larger negative offset values the idea is the same.

3.4.4. Set–Based Graphs

Set–Based Graphs are implemented using the *Boost Graph Library* [18] defining the corresponding `SetVertex` and `SetEdge` data structures as *Vertex* and *Edge* properties of the graph implementation. That way, the graph is defined by a list of n `SetVertexes` and m `SetEdges`.

Every set-edge contains a domain set (representing edges) and two maps, `SetEdge.map1` and `SetEdge.map2`, that compute the vertices connected by each edge in the domain set. For instance, the edge $e \in$`SetEdge.domain` connects the vertices $v_1 =$`SetEdge.map1`(e) and $v_2 =$`SetEdge.map2`(e).

On this class, we implemented the function `connectComp` that computes the connected components of a given SB-Graph. This function returns a map D_{map} as explained in Section 3.2.

3.4.5. Implementation Restrictions

While Algorithm 2 is general, the implementation described above imposes the following restrictions on the set–based graphs:

1. Every individual vertex is represented by an array of natural numbers of dimension d.

2. Every set-vertex is a union of a finite number of atomic sets represented by intervals of dimension d. Recall that every interval in each dimension is defined by three natural numbers: *start*, *step*, and *end*.

3. The maps that define the set edges $\text{map}^{h,i} : \mathbb{N}^d \to \mathbb{N}^d$ are *piecewise linear*. Each map has a finite number of domains with a corresponding linear affine function. In every domain, the function acting in each dimension is characterized by two rational numbers: the *gain* and the *offset*.

4. The implementation of the mapInf function imposes a further restriction to the maps: In a given domain and dimension, if $\text{map}^{h,i}$ and $\text{map}^{h,j}$ have both nonzero gains, then the gains must be the same. Otherwise, function minAdjMap might return a map with some gain that is not 1 or 0 and, if that map turns to be greater than the identity, then mapInf cannot be applied.

The last restriction can be easily avoided with a more general implementation of mapInf considering gains different from 1 and 0.

4. Efficient Connection Flattening

In this section we describe the methodology that, making use of the algorithm introduced in Section 3, allows replacing connections by equations in object oriented models.

4.1. General Procedure

In order to automatically obtain a code like that of Listing 2 given a set of connections like those of Listing 1, we propose the following procedure:

1. Build a SB Graph:
   - Associate a set-vertex to each array of connectors. For the example of Listing 1 the arrays of connectors are $S.p$, $S.n$, $G.p$, $R[1:N].p$, $R[1:N].n$, $C[1:N].p$, and $R[1:N].n$. Then, each array is associated to a set-vertex where each individual vertex can be represented in this case by a natural number (because the problem is one-dimensional).

     The association between individual connectors and vertices can be represented by a function $v = \text{vertex}(x)$ denoting that vertex v is associated to connector x.
   - Associate a set-edge to each set of connections between every pair of set vertices. In the example some set edges would be
     - $E^1 = E^1[S.p, R.p]$, characterized by $\text{map}_1^1(e_1^1) = \text{vertex}(S.p)$ and $\text{map}_2^1(e_1^1) = \text{vertex}(R[1].p)$.
     - $E^2 = E^2[R.n, R.p]$, characterized by $\text{map}_1^2(e_i^2) = \text{vertex}(R[i].n)$ and $\text{map}_2^2(e_i^2) = \text{vertex}(R[i+1].p)$ for $i = 1, \ldots, N-1$.
     - $E^3 = E^3[C.n, G.p]$, characterized by $\text{map}_1^3(e_i^3) = \text{vertex}(C[i].n)$ and $\text{map}_2^3(e_i^3) = \text{vertex}(G.p)$ for $i = 1, \ldots, N$.
2. Find the connected components of the SB–Graph using Algorithm 3.
3. Given the map D_{map} representing the sets of connected components, write the corresponding equations. A possible way of doing it is using Algorithm 4.

Algorithm 4 Code Generation Procedure

```
function WriteCode($D_{\text{map}}$)
    $I_k \leftarrow$ split(Image[$D_{\text{map}}$])    ▷ Split image set into atomic sets
    for all $I_k \subseteq$ Image[$D_{\text{map}}$] do
        $\hat{D}_k \leftarrow D_{\text{map}}^{-1}(I_k) \setminus D_{\text{map}}(D_k)$    ▷ Set of vertices connected to $I_k$
        $\hat{D}^k_j \leftarrow$ split($\hat{D}_k$)    ▷ Split domain vertices into atomic sets. The fact
    that $\hat{D}^k_j$ and $I_k$ are atomic sets implies that $D_{\text{map}}$ has the same expression
    for all $i \in D^k_j$.
        for all $D^k_j \subseteq \hat{D}_k$ do
            Code ← Code+"for i in interval($\hat{D}^k_j$) loop"
            Code ← Code+" ef[i] = ef[$D_{\text{map}}$(i)];"
            Code ← Code+"end for;"
        end for
        Code ← Code+" for i in interval($I^k$) loop"
        Code ← Code+" fl[i]"
        for all $D^k_j \subseteq \hat{D}_k$ do
            Code ← Code+"+ sum(fl[i1], for i1 in interval($\hat{D}^k_j$)) "
        end for
        Code ← Code+"= 0; "    ▷ End of line with sum of flows equal to 0.
        Code ← Code+"end for;"
    end for
    return Code
end function
```

Algorithm 4 first splits the domain and image sets of D_{map} into atomic sets, i.e., sets that can be represented by simple intervals and where D_{map} has a simple expression of the type $a \cdot i + b$. Then, it generates the effort equations inside `for loop` statements that traverse the domain intervals and replace D_{map} by its expression in the index at the right hand side of each equation. That way, the compact code produced generates an equation for each element of the domain saying that the corresponding effort is equal to that of its representative in the connected component.

Then, it proceeds in a similar way with the flow equation, but this time the `for loop` statements traverse the image of D_{map} (i.e., the representatives of each connected component) generating an equation for each connected component that contains the sum of the flows of all its elements.

In presence of multidimensional arrays the indices `i` and `i1` are simply replaced by multidimensional expressions like i, j, k.

Algorithm 4 writes a model in terms of global flow and effort arrays (`fl` and `ef`, respectively). These arrays must be then replaced by the actual names of the connector variables.

4.2. Analysis of the Restrictions

The restrictions described in Sec.3.4.5 about the implementation and the conditions enumerated after Lemma 1 establish the circumstances under which the algorithm effectively achieves a constant cost with respect to the number of vertices and edges. While these conditions may be quite restrictive in general, in the context of replacing connections by equations in object oriented models they are almost invariantly satisfied:

- The connectors in a model are always instantiated as scalar or arrays with different dimensions. We can represent all of them using arrays of vertices with the maximum dimension found. That way the first two restrictions of Sec.3.4.5 are always satisfied.
- The third restriction is satisfied provided that:
  - In presence of nested `for loop` statements, the interval of the iterators are independent on each other. This is, we cannot write `for i in 1:N loop; for j in 1:i loop` since in that case the domain of the maps defining the set edges would not be an interval.
  - The connections have linear affine operations with each index. This is, we can only have expressions like `connect(v[a*i+b, c*j+d], w[e*i+f, g*j+h])` where `i` and `j` are the nested iterators and `a, b, c, d, e, f, g, h` are rational constants.
- The fourth restriction is satisfied provided that `a` and `e` in the previous item are different only if one of them is zero (and the same for `c` and `g`).

Regarding the conditions listed after Lemma 1 under which the algorithm performs a limited number of iterations, they are automatically satisfied under the assumption that the maps are piece-wise linear since in that case any large set of connected connectors will keep a strict ordering.

5. Examples and Results

We introduce next three examples where we applied the presented methodology implemented in the ModelicaCC prototype compiler.

We compared the CPU time taken by the flattening process of our implementation with that of the algorithm used by OpenModelica. Furthermore, in order to corroborate that the flattened model generated by ModelicaCC is correct, we simulated it using OpenModelica and compared the results with those of the original model checking that the difference between the state trajectories of both models was negligible.

In all the models, and for the different size settings, we used the following experimental pipeline:

1. Generate a flattened Modelica model using the algorithm implemented in ModelicaCC.

2. Generate a flattened Modelica model using OpenModelica (using the *-s* flag) and then compare the flattened model generation time of both tools.
3. Compile the generated model in step 1 with OpenModelica to obtain an executable model.
4. Compile the original model with OpenModelica to obtain an executable model.
5. Simulate both models with a small tolerance (10^{-10}) and compute the difference between both simulation results using the *Normalized Mean Square Error* metric on all state variables (this sanity check was only done in small sized models).

We run the experiments with an Intel i9 core with 32 GB of RAM memory running Ubuntu 20.04 OS[2].

5.1. Simple RC Network

We consider first the example of Listing 1 with $N = 100$. In this example, ModelicaCC associates the original Modelica connectors with the corresponding vertices in the resulting SBG as shown in Table 1. The SBG is represented in Figure 2.

Connector	**Vertex**
$C[i].n$	i
$C[i].p$	$i + 100$
$G.p$	201
$R[i].n$	$i + 201$
$R[i].p$	$i + 301$
$S.n$	402
$S.p$	403

Table 1: SB Graph vertices and connectors association for the RC network (for $i \in [1 : 1 : 100]$).

Then, using Algorithm 3, the map D_{map} results as follows:

[2]All the models presented in this article and a complete description on how to apply the flatter algorithm can be found at: `https://github.com/CIFASIS/modelicacc/tree/v3.0/test/mccprograms/flatter`

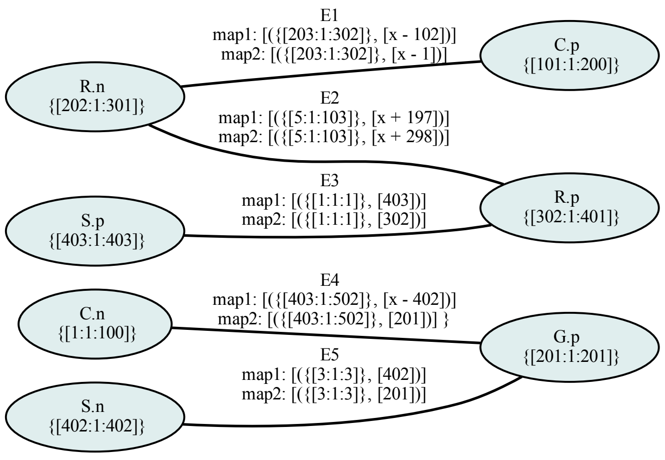

E1
map1: [({[203:1:302]}, [x - 102])]
map2: [({[203:1:302]}, [x - 1])]
C.p
{[101:1:200]}
R.n
{[202:1:301]}
E2
map1: [({[5:1:103]}, [x + 197])]
map2: [({[5:1:103]}, [x + 298])]
E3
map1: [({[1:1:1]}, [403])]
map2: [({[1:1:1]}, [302])]
S.p
{[403:1:403]}
R.p
{[302:1:401]}
E4
map1: [({[403:1:502]}, [x - 402])]
map2: [({[403:1:502]}, [201])] }
C.n
{[1:1:100]}
G.p
{[201:1:201]}
E5
map1: [({[3:1:3]}, [402])]
map2: [({[3:1:3]}, [201])]
S.n
{[402:1:402]}

Figure 2: RC network generated graph.

$$D_{\text{map}}(i) = \begin{cases} i & \text{if } i \in \{1:1:1\} \\ 1 & \text{if } i \in \{2:1:100\} \\ i & \text{if } i \in \{101:1:199\} \\ i & \text{if } i \in \{200:1:200\} \\ 1 & \text{if } i \in \{201:1:201\} \\ i-101 & \text{if } i \in \{202:1:300\} \\ i-101 & \text{if } i \in \{301:1:301\} \\ 302 & \text{if } i \in \{302:1:302\} \\ i-202 & \text{if } i \in \{303:1:401\} \\ 1 & \text{if } i \in \{402:1:402\} \\ 302 & \text{if } i \in \{403:1:403\} \end{cases}$$

which can be easily verified to be correct. For instance, the fact that $D_{\text{map}}(i) = 1$ for $2 \leq i \leq 100$, for $i = 201$ and for $i = 402$ implies that the vertex 1 represents the connected component formed by those values of i. Recalling the association between connectors and vertices in Table 1, the vertex associated to connector $C[1].n$ is the representative of the connected components formed by $C[i].n$ $(1 \leq i \leq 100)$, $G.p$ and $S.n$.

We then changed the size of the model from $N = 100$ to $N = 20000$ measuring the CPU time taken by ModelicaCC and OpenModelica to produce the flattened model. Table 2 reports the results showing that, as expected, ModelicaCC has constant computational costs while in OpenModelica the CPU time grows more than linearly for large values of N.

	SBG Implementation	OpenModelica
Size	**Flattening Time [msec]**	**Flattening Time [msec]**
100	26	107
200	27	153
500	27	395
1000	26	696
2000	25	1235
5000	27	3905
10000	27	9390
20000	25	25481

Table 2: Simple RC Network results with different size of parameter N

Listing 3 exhibits part of the flattened model produced by ModelicaCC for $N = 100$. It can be seen that it preserves the repetitive structures generating a compact model.

Listing 3: Generated Equations for RC Network model with N

```
model RC
  ...
equation
  ...
  for i in 1:1:99 loop
    C_p_i[i]+R_p_i[i+1]+R_n_i[i] = 0;
  end for;
  C_p_i[100]+R_n_i[100] = 0;
  C_n_v[1] = G_p_v;
  C_n_v[1] = S_n_v;
  for i in 1:1:99 loop
    C_n_v[1] = C_n_v[i+1];
  end for;
  C_n_i[1]+G_p_i+S_n_i+sum(C_n_i[2:1:100]) = 0;
  R_p_v[1] = S_p_v;
  R_p_i[1]+S_p_i = 0;
  C_p_v[100] = R_n_v[100];
  for i in 1:1:99 loop
    C_p_v[i] = R_p_v[i+1];
    C_p_v[i] = R_n_v[i];
  end for;
end RC;
```

5.2. RC Network with Recursive Connection

For the same system of Figure 1, we rewrote the model connections as described in Listing 4.

Listing 4: Modelica connections

```
connect(S.p,R[1].p);
connect(S.n,G.p);
connect(C[1].n,G.p);
for i in 1:N-1 loop
  connect(R[i].n, R[i+1].p);
  connect(C[i+1].n, C[i].n); //recursive connection
```

```
end for;
for i in 1:N loop
  connect(C[i].p, R[i].n);
end for;
```

ModelicaCC associated the connectors and SBG vertices as in the previous case (Table 1) but now the algorithm finds the following map of connected components:

$$D_{\mathrm{map}}(i) = \begin{cases} 1 & \text{if } i \in \{1:1:1\} \\ 1 & \text{if } i \in \{2:1:2\} \\ 1 & \text{if } i \in \{3:1:99\} \\ 1 & \text{if } i \in \{100:1:100\} \\ i & \text{if } i \in \{100:1:199\} \\ i & \text{if } i \in \{200:1:200\} \\ 1 & \text{if } i \in \{201:1:201\} \\ i-101 & \text{if } i \in \{202:1:300\} \\ i-101 & \text{if } i \in \{301:1:301\} \\ 302 & \text{if } i \in \{302:1:302\} \\ i-202 & \text{if } i \in \{303:1:401\} \\ 1 & \text{if } i \in \{402:1:402\} \\ 302 & \text{if } i \in \{403:1:403\} \end{cases}$$

The map, as expected, is exactly the same as before, but it is now described using more sub-intervals. This is caused by the usage of the `mapInf` function that solves the recursive connection on $C.n$ without iterating.

Like in the previous case, we changed the size of the model from $N = 100$ to $N = 20000$ measuring the CPU time taken by ModelicaCC and OpenModelica to produce the flattened model. Table 3 reports the results that are very similar to the previous ones.

	SBG Implementation	**OpenModelica**
Size	**Flattening Time [msec]**	**Flattening Time [msec]**
100	47	104
200	48	163
500	47	370
1000	48	633
2000	47	1250
5000	49	4180
10000	47	9450
20000	49	26927

Table 3: Recursive RC Network system results with different size of parameter N

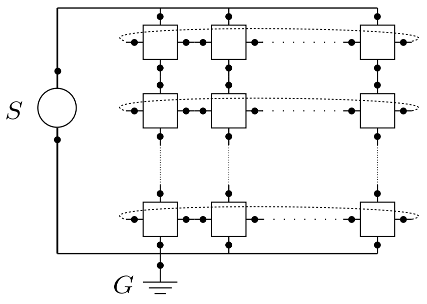
Listing 5 exhibits part of the flattened model produced by ModelicaCC for $N = 100$. The fact that the intervals are more partitioned than before produces a slightly longer piece of code.

Listing 5: Generated Equations for RC Network

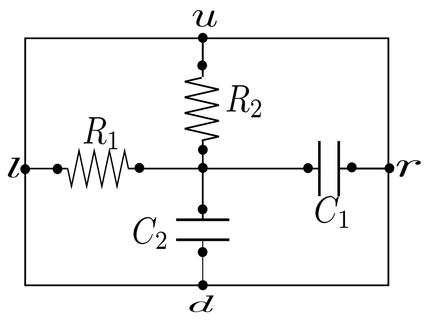
```
model RecursiveRC
  ...
equation
  ...
  C_n_v[1] = G_p_v;
  C_n_v[1] = C_n_v[2];
  C_n_v[1] = S_n_v;
  C_n_v[1] = C_n_v[100];
  for i in 1:1:97 loop
    C_n_v[1] = C_n_v[i+2];
  end for;
  C_n_i[1]+G_p_i+sum(C_n_i[3:1:99])+C_n_i[2]+S_n_i+C_n_i[100] = 0;
  for i in 1:1:99 loop
    C_p_i[i]+R_n_i[i]+R_p_i[i+1] = 0;
  end for;
  C_p_i[100]+R_n_i[100] = 0;
  R_p_v[1] = S_p_v;
  R_p_i[1]+S_p_i = 0;
  C_p_v[100] = R_n_v[100];
  for i in 1:1:99 loop
    C_p_v[i] = R_n_v[i];
    C_p_v[i] = R_p_v[i+1];
  end for;
end RecursiveRC;
```

5.3. A Two-Dimensional RC Network

The third example consists of a 2D network formed by $N \times M$ cells with 4 connectors each (left, right, up and down connectors), a ground component with one connector and a source component with two connectors. The network is connected as it is shown in Figure 3a and expressed in Listing 6. Each cell internally contains two capacitors and two resistors connected as shown in Figure 3b and described in Listing 7.



(a) 2D Network



(b) 2D Network Cell

Listing 6: Modelica connections

```
for i in 1:N-1,j in 1:M-1 loop
  connect(Cell[i,j].r, Cell[i,j+1].l);
  connect(Cell[i,j].d, Cell[i+1,j].u);
end for;
for i in 1:N loop
  connect(Cell[i,M].r, Cell[i,1].l);
end for;
for j in 1:M loop
  connect(Cell[1,j].u,S.p);
  connect(Cell[N,j].d,S.n);
end for;
```

Listing 7: Modelica connections for each cell

```
connect(R1.n, C1.p);
connect(R2.n, C2.p);
connect(R2.n, C1.p);
connect(R1.p, l);
connect(C1.n, r);
connect(R2.p, u);
connect(C2.n, d);
```

For this model, with $N = M = 100$, ModelicaCC constructs the SBG associating connectors and vertices as shown in Table 4.

Connector	**Vertex**
$Cell[i,j].C2.n$	(i,j)
$Cell[i,j].C2.p$	$(i+100, j+100)$
$Cell[i,j].C1.n$	$(i+200, j+200)$
$Cell[i,j].C1.p$	$(i+300, j+300)$
$Cell[i,j].d$	$(i+400, j+400)$
$Cell[i,j].l$	$(i+500, j+500)$
$Cell[i,j].r$	$(i+600, j+600)$
$Cell[i,j].R2.n$	$(i+700, j+700)$
$Cell[i,j].R2.p$	$(i+800, j+800)$
$Cell[i,j].R1.n$	$(i+900, j+900)$
$Cell[i,j].R1.p$	$(i+1000, j+1000)$
$Cell[i,j].u$	$(i+1100, j+1100)$
$G.p$	$(1202, 1202)$
$S.n$	$(1203, 1203)$
$S.p$	$(1204, 1204)$
$-Cell[i,j].d$	$(i+1203, j+1203)$
$-Cell[i,j].l$	$(i+1303, j+1303)$
$-Cell[i,j].r$	$(i+1403, j+1403)$
$-Cell[i,j].u$	$(i+1503, j+1503)$

Table 4: SB Graph vertices and connectors association for the 2D network.

On the resulting graph, the proposed algorithm finds the connected components represented by the following map:

$$D_{\text{map}}(i,j) = \begin{cases} (i,j) & \text{if } (i,j) \in \{[1:1:100] \times [1:1:100]\} \\ (i,j) & \text{if } (i,j) \in \{[101:1:200] \times [101:1:200]\} \\ (i,j) & \text{if } (i,j) \in \{[201:1:300] \times [201:1:300]\} \\ (i-200, j-200] & \text{if } (i,j) \in \{[301:1:400] \times [301:1:400]\} \\ (i,j) & \text{if } (i,j) \in \{[401:1:499] \times [401:1:499]\} \\ (i,j) & \text{if } (i,j) \in \{[401:1:499] \times [500:1:500]\} \\ (500,401) & \text{if } (i,j) \in \{[500:1:500] \times [401:1:401]\} \\ (500,401) & \text{if } (i,j) \in \{[500:1:500] \times [402:1:500]\} \\ (i,j) & \text{if } (i,j) \in \{[501:1:599] \times [502:1:600]\} \\ (i,j+501) & \text{if } (i,j) \in \{[501:1:600] \times [501:1:501]\} \\ (i,j) & \text{if } (i,j) \in \{[600:1:600] \times [502:1:600]\} \\ (i-100, j-99) & \text{if } (i,j) \in \{[601:1:699] \times [601:1:699]\} \\ (i-100, j+501) & \text{if } (i,j) \in \{[601:1:700] \times [700:1:700]\} \\ (i,j) & \text{if } (i,j) \in \{[700:1:700] \times [601:1:609]\} \\ (i-600, j-600) & \text{if } (i,j) \in \{[701:1:800] \times [701:1:800]\} \\ (i,j) & \text{if } (i,j) \in \{[801:1:900] \times [801:1:900]\} \\ (i-800, j-800) & \text{if } (i,j) \in \{[901:1:1000] \times [901:1:1000]\} \\ (i,j) & \text{if } (i,j) \in \{[1001:1:1100] \times [1001:1:1100]\} \\ (1101,1101) & \text{if } (i,j) \in \{[1101:1:1101] \times [1101:1:1101]\} \\ (1101,1101) & \text{if } (i,j) \in \{[1101:1:1101] \times [1102:1:1200]\} \\ (i-701, j-700) & \text{if } (i,j) \in \{[1102:1:1200] \times [1101:1:1199]\} \\ (i,j) & \text{if } (i,j) \in \{[1102:1:1200] \times [1200:1:1200]\} \\ (500,401) & \text{if } (i,j) \in \{[1202:1:1202] \times [1202:1:1202]\} \\ (1101,1101) & \text{if } (i,j) \in \{[1203:1:1203] \times [1203:1:1203]\} \\ (i-203, j-203) & \text{if } (i,j) \in \{[1204:1:1303] \times [1204:1:1303]\} \\ (i-1103, j-1103) & \text{if } (i,j) \in \{[1304:1:1403] \times [1304:1:1403]\} \\ (i-603, j-603) & \text{if } (i,j) \in \{[1404:1:1503] \times [1404:1:1503]\} \\ (i-1503, j-1503) & \text{if } (i,j) \in \{[1504:1:1603] \times [1504:1:1603]\} \end{cases}$$

that can be also verified to be correct. Like in the previous examples, we changed the values of N and M and measured the flattening time taken by ModelicaCC and OpenModelica. As expected, the SBG based approach achieved a constant time.

Listing 8 shows part of the flattened model produced by ModelicaCC for $N = M = 100$.

Listing 8: Generated Equations for 2D Network

```
model N2D
  ...
equation
```

Size	SBG Implementation	OpenModelica
	Flattening Time [msec]	Flattening Time [msec]
10x10	38	191
20x20	38	705
50x50	37	5210
100x100	38	18812
200x200	37	70810

Table 5: Two-Dimensional Network results with different grid size parameter N

```
...
for i in 1:1:99,j in 1:1:99 loop
  Cell_d_i[i,j]+Cell_u_i[i+1,j] = 0;
end for;
Cell_u_v[1,1] = S_p_v;
for i in 1:1:1,j in 1:1:99 loop
  Cell_u_v[1,1] = Cell_u_v[1,j+1];
end for;
Cell_u_i[1,1]+S_p_i+sum(Cell_u_i[1:1:1, 2:1:100]) = 0;
Cell_d_v[100,1] = S_n_v;
for i in 1:1:1,j in 1:1:99 loop
  Cell_d_v[100,1] = Cell_d_v[100,j+1];
end for;
Cell_d_i[100,1]+S_n_i+sum(Cell_d_i[100:1:100, 2:1:100]) = 0;
for i in 1:1:100,j in 1:1:100 loop
  Cell_R1_p_i[i,j]+(-Cell_l_i[i,j]) = 0;
end for;
for i in 1:1:100,j in 1:1:100 loop
  Cell_C1_n_i[i,j]+(-Cell_r_i[i,j]) = 0;
end for;
for i in 1:1:100,j in 1:1:100 loop
  Cell_R2_p_i[i,j]+(-Cell_u_i[i,j]) = 0;
end for;
for i in 1:1:100,j in 1:1:100 loop
  Cell_C2_n_i[i,j]+(-Cell_d_i[i,j]) = 0;
end for;
for i in 1:1:99,j in 1:1:99 loop
  Cell_l_i[i,j+1]+Cell_r_i[i,j] = 0;
end for;
for i in 1:1:100,j in 1:1:1 loop
  Cell_l_v[i,1] = Cell_r_v[i,100];
end for;
for i in 1:1:100,j in 1:1:1 loop
  Cell_l_i[i,1]+Cell_r_i[i,100] = 0;
end for;
for i in 1:1:100,j in 1:1:100 loop
  Cell_C2_p_i[i,j]+Cell_R2_n_i[i,j]+Cell_C1_p_i[i,j]+Cell_R1_n_i[i,j] =
      0;
end for;
for i in 1:1:99,j in 1:1:1 loop
  Cell_u_i[i+1,100] = 0;
end for;
for i in 1:1:99,j in 1:1:1 loop
  Cell_d_i[i,100] = 0;
end for;
for i in 1:1:1,j in 1:1:99 loop
  Cell_l_i[100,j+1] = 0;
end for;
for i in 1:1:1,j in 1:1:99 loop
  Cell_r_i[100,j] = 0;
  end for;
  G_p_i = 0;
  for i in 1:1:100,j in 1:1:100 loop
    Cell_C2_p_v[i,j] = Cell_R2_n_v[i,j];
    Cell_C2_p_v[i,j] = Cell_C1_p_v[i,j];
    Cell_C2_p_v[i,j] = Cell_R1_n_v[i,j];
  end for;
  for i in 1:1:99,j in 1:1:99 loop
    Cell_l_v[i,j+1] = Cell_r_v[i,j];
  end for;
  for i in 1:1:100,j in 1:1:100 loop
    Cell_C2_n_v[i,j] = Cell_d_v[i,j];
  end for;
  for i in 1:1:100,j in 1:1:100 loop
    Cell_R2_p_v[i,j] = Cell_u_v[i,j];
  end for;
  for i in 1:1:100,j in 1:1:100 loop
    Cell_C1_n_v[i,j] = Cell_r_v[i,j];
  end for;
  for i in 1:1:100,j in 1:1:100 loop
    Cell_R1_p_v[i,j] = Cell_l_v[i,j];
  end for;
  for i in 1:1:99,j in 1:1:99 loop
    Cell_d_v[i,j] = Cell_u_v[i+1,j];
  end for;
end N2D;
```

6. Conclusions and Future Research

We presented a novel methodology for replacing large sets of connections by the corresponding equations in object-oriented modeling languages. The procedure makes use of a new algorithm for finding connected components in undirected graphs that, under certain regularity assumptions, has constant computational costs with the number of vertices and edges. This is achieved using the concept of *Set-Based Graphs* and, to the best of our knowledge, constitutes the first algorithm of this type.

We described also the implementation of the proposed methodology as part of flattening stage of the ModelicaCC compiler. In addition, we demonstrated the usefulness and the functionality of the algorithm through three examples of large scale graphs, including a two-dimensional case.

We believe this work opens several future lines of work and research. We are currently working on developing more algorithms of this type (using SB-Graphs with maps) for other problems related to Modelica compilation: finding maximum matching in bipartite graphs and strongly connected components (directed graphs). Some algorithms for these problems were already proposed using SB-Graphs in [20], but those algorithms only worked for particular cases and we believe that they can be generalized exploiting some concepts developed in this work such as using maps for representing set-edges.

Besides these new problems, there are several issues related to the algorithm presented here that should be taken into account in the future. Among them, it would be important to establish some bounds on the cost of every step of the algorithm with respect to the number of different linear maps that are used to describe each map. In addition, we need to find less restrictive conditions under which the algorithm actually has a constant cost with respect to the size of the sets.

Another important goal is that of implementing the methodology in a more robust and complete Modelica compiler such as OpenModelica [9] imposing also less restrictive conditions on the set representation. The current implementation only allows the use of diagonal linear affine maps which may limit the usability of the methodology in some practical cases.

Finally, we believe that this algorithm can be effectively applied in other fields beyond object oriented models. Any problem leading to analysis on a large graph containing some regular connections is in principle a good candidate to be solved using SB-Graphs.

Funding

This work was partially funded by grant PICT–2017 2436 (ANPCYT).

References

[1] Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, number 157. Linköping University Electronic Press, 2019.

[2] Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. Jmodelica—an open source platform for optimization of modelica models. In *6th Vienna International Conference on Mathematical Modelling*, 2009.

[3] 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*, pages 35–38, 2014.

[4] EC Federico Bergero, Mariano Botta, and Ernesto Kofman. Efficient compilation of large scale modelica models. In *11th International Modelica Conference*, 2015.

[5] Willi Braun, Francesco Casella, Bernhard Bachmann, et al. Solving large-scale modelica models: new approaches and experimental results using openmodelica. In *12 International Modelica Conference*, pages 557–563. Linkoping University Electronic Press, 2017.

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

[7] Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *11th International Modelica Conference*, pages 459–468, 2015.

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

[9] Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. Openmodelica-a free open-source environment for system modeling, simulation, and teaching. In *2006 IEEE Conference on Computer Aided Control System Design*, pages 1588–1595. IEEE, 2006.

[10] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system modeling and simulation. In *European Conference on Object-Oriented Programming*, pages 67–90. Springer, 1998.

[11] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[12] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[13] Ernesto Kofman, Joaquín Fernández, and Denise Marzorati. Compact sparse symbolic jacobian computation in large systems of odes. *Applied Mathematics and Computation*, 403:126181, 2021.

[14] Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, and Rüdiger Franke. A new openmodelica compiler high performance frontend. In *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, number 157. Linköping University Electronic Press, 2019.

[15] Xiaolin Qin, Juan Tang, Yong Feng, Bernhard Bachmann, and Peter Fritzson. Efficient index reduction algorithm for large scale systems of differential algebraic equations. *Applied Mathematics and Computation*, 277:10–22, 2016.

[16] Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E Nagel, and Ines Gubsch. Exploiting repeated structures and vectorization in modelica. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, number 118, pages 265–272. Linköping University Electronic Press, 2015.

[17] Gerald Schweiger, Henrik Nilsson, Josef Schoeggl, Wolfgang Birk, and Alfred Posch. Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms. *Applied Mathematics and Computation*, 365:124713, 2020.

[18] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual.* Addison-Wesley, 2002.

[19] Kristian Stavåker. *Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures.* PhD thesis, Linköping University Electronic Press, 2015.

[20] Pablo Zimmermann, Joaquín Fernández, and Ernesto Kofman. Set-based graph methods for fast equation sorting in large dae systems. In *Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools*, pages 45–54, 2019.