# Constant Time Causalization using Resizable Arrays

Karim Abdelhak[1] Bernhard Bachmann[1]

[1]Institute for Data Science Solutions, Bielefeld University of Applied Sciences and Arts, Germany,
{karim.abdelhak, bernhard.bachmann}@hsbi.de

## Abstract

Equation-based modeling that utilizes reusable components to represent real-world systems can result in excessively large models. This, in turn, significantly increases compilation time and code size, even when employing state-of-the-art scalarization and causalization techniques. This paper presents an algorithm that leverages repeating patterns and uniform causalization to enable array-size-independent constant time processing. Allowing structural parameters that govern array sizes to remain resizable during and after the causalization process enables the formulation of an integer-valued nonlinear optimization problem. This approach identifies the minimal model configuration that preserves the required structural integrity, which can subsequently be resized as needed for simulation. The proposed method has been implemented in OpenModelica and builds upon preliminary work aimed at preserving array structures during causalization, while still resolving the underlying problem in a scalarized manner.

*Keywords: Equation-based modelling, Array-preserving, Causalization, Resizable, Nonlinear programming, Integer programming*

## 1 Introduction

Current state-of-the-art approaches for compiling large-scale models with scalable structures typically rely on full scalarization. Although this method yields correct results, it often incurs substantial compilation times and leads to significantly larger code sizes. *OpenModelica*'s (Fritzson, Pop, Abdelhak, et al. 2020) pseudo-array compilation algorithm (Abdelhak, Casella, and Bachmann 2023) addresses the challenges of large code sizes and prolonged compilation times by enabling array-size-independent symbolic manipulation, while still performing causalization on scalarized graphs. Building on this foundation, this paper presents an algorithm that systematically reduces scalable structures to a theoretical minimal form, allowing causalization to be performed efficiently. Following compilation, the resulting model can be resized to any desired scale suitable for simulation.

Thematically related research exploring the exploitation of resizable array structures includes the work presented in (Neumayr and Otter 2023), which introduces predefined acausal components as a means of enabling flexible array configurations. Additionally, an alternative to the pseudo-array compilation algorithm has been proposed in (Marzorati, Fernández, and Kofman 2024), where a set-based graph approach is employed. This method is inherently designed to operate independently of array sizes, representing a structurally distinct approach to addressing the challenges associated with scalable model compilation. Although the paper does not explicitly address the resizability of set-based nodes within the set-based graph, the algorithms presented in this paper for determining the general resizability of for-loop equations should be equally applicable, given that the underlying model remains unchanged.

### 1.1 Resizable Parameters

First and foremost, structural parameters that determine the array sizes within a model must be preserved and not evaluated throughout the flattening process. These parameters are *resizable* but will be simply referred to as *parameters* in all subsequent discussions. While the concept of minimizing model size may appear to align directly with minimizing the values of these parameters, such a correlation is not guaranteed. The modeling language *Modelica* permits array sizes to be defined by any expression that yields a non-negative integer (Modelica Association 2023). As a result, the relationship between parameter values and overall model size can even be nonlinearly correlated. Additionally, a single parameter may influence multiple array sizes, and conversely, multiple parameters may contribute to determining a single array size. *OpenModelica* offers two different ways to indicate that a parameter is *resizable*, which will be described in detail in Section 3.3.

### 1.2 Resizable Equations

Any equation whose size is determined by a resizable parameter is itself considered resizable. In the modeling language *Modelica*, there are two primary constructs for expressing such equations: *array equations* and *for-equations*. While array equations are a conventional method for representing multidimensional relationships, the latter are more specific to Modelica. For-equations employ a surrounding for-loop construct—similar to those used in algorithmic programming—to define a set of structurally similar relationships using array variables indexed by one or more iterator variables.

The primary advantage of identifying resizable equations lies in the ability to reduce their size as much as pos-

sible during the process of causalization. This enables the transformation of the flattened model into target code to be performed independently of the resizable array sizes defined within the model. It is important to note that in order to keep the system structure parameters and equations must satisfy specific conditions and cannot be resized arbitrarily.

**Assumption 1** *The following discussion operates under the assumption that the model remains balanced for every feasible combination of parameter values. In other words, each parameter is assumed to influence variable dimensions and corresponding equation sizes in a consistent manner.*

**Example 1** *As an example of an invalid resizable parameter, consider the following model. While the original definition with $p = 2$ yields a balanced and valid model, any other value of p results in an invalid configuration.*

```
model invalid_resizable
  parameter Integer p = 2;
  Real[p*p] x
equation
  for i in 1:2*p loop
    x[i] = sin(time) * i;
  end for;
end invalid_resizable;
```

# 2 The Optimization Problem

The primary objective of the optimization is to minimize the overall model size. Owing to Assumption 1, this objective can be simplified to minimizing either the total number of variables or equations, as their sizes are assumed to be proportionally aligned. In the following, the optimization problem is formulated with a focus on minimizing the sizes of the equations. As will be demonstrated in the following sections, the decision variables are integer-valued, and the objective function is inherently nonlinear. The constraint set includes both inequalities and equalities, which—although infrequently—may also exhibit nonlinear characteristics.

## 2.1 Objective Function

Let $x \in \mathbb{Z}^n$ denote the vector of decision variables representing resizable parameters. The goal is to determine the values of $x$ that minimize the total number of equations, subject to structural and semantic constraints.

$$\arg\min_{x} \quad F(x) = \sum_{k} f_k(x) \tag{1}$$

where $F(x)$ is the total model size and $f_k$ being the local size objectives of equation $k$ as product of their dimension objective functions:

$$f_k(x) = \prod_{j} d_j^k(x) \tag{2}$$

The dimension objective function evaluates whether the $j$-th dimension $D$ satisfies condition $(a_j)$, namely, that it is defined by an iterator within a for-loop whose range $D_{\text{start}} : D_{\text{step}} : D_{\text{stop}}$ is governed by one or more resizable parameters. If $(a_j)$ holds, the objective is set to the number of elements traversed by the iterator. Otherwise, if the size is not explicitly defined by an iterator, the equation must be an array-based equation whose dimensions can be inferred from the array variables it contains. This inference is performed by the function $\dim_j^k(x)$ which may either yield a constant size or, in the case of parameterized array equations, represent a function dependent on one or more resizable parameters.

$$d_j^k(x) = \begin{cases} (D_{\text{stop}}(x) - D_{\text{start}}(x))/D_{\text{step}} + 1 & \text{if } (a_j) \\ \dim_j^k(x) & \text{else} \end{cases} \tag{3}$$

## 2.2 Constraints

This optimization problem is subject to several constraints, which can be classified into three main categories:

**equation dimension constraints.** The first set of constraints arises from the requirement that each resizable equation must have a size of at least one to ensure its representation in the underlying bipartite graph. Consequently, each dimension associated with an equation must be of minimum size one. For every iterator $i$ with a range defined as $D^i_{\text{start}} : D^i_{\text{step}} : D^i_{\text{stop}}$, this requirement can be expressed through the following constraint:

$$0 \leq (D^i_{\text{stop}}(x) - D^i_{\text{start}}(x))/D^i_{\text{step}} \tag{$g_1$}$$

Furthermore, the same condition must hold for each $j$-th dimension of the $k$-th array equation that depends on resizable parameters imposing the following constraint:

$$1 \leq \dim_j^k(x) \tag{$g_2$}$$

**structural variable constraints.** For each resizable parameter $x_p \in x$ the minimum and maximum value constraints (if given by the underlying model) have to be taken into account:

$$x_p \geq x_p^{\min} \tag{$g_3$}$$
$$x_p \leq x_p^{\max} \tag{$g_4$}$$

In addition to being box-constrained by their minimum and maximum values, parameters may also impose mutual restrictions. When a resizable variable is indexed using a resizable parameter, it is essential to ensure that this indexing operation does not result in an out-of-range access.

**Example 2** *Let y be a resizable variable whose size is determined by the parameter $x_1$. Additionally, consider a component reference to y in the model as*


DOI 10.3384/ecp218203

*$y[x_2]$, where $x_2$ is another parameter. This imposes a constraint on $x_2$, requiring that $x_2 \leq x_1$ to prevent out-of-range access.*

The same logic applies to non-parameter indexed variables, constant valued restrictions have to be collected as well. For each component reference that references a resizable variable in the model, each dimension has to be checked if it imposes a constraint.

$$\text{sub}_c^j(x) \leq \text{cdim}_c^j(x) \qquad (g_5)$$

where $c$ refers to $c$-th component reference and $j$ to the $j$-th dimension $\text{cdim}_c^j(x)$ with $\text{sub}_c^j(x)$ being the subscript of $c$ in dimension $j$. If a constraint does not involve any parameters or if the parameters cancel out, its validity must be verified. If the constraint is valid, it can be safely omitted; otherwise, an error should be raised, as this indicates an issue with the underlying model.

**structural equation constraints.** The final set of constraints is derived from array equations and, in contrast to the previous inequality constraints, consists of equality constraints. For each array equation $k$, the $j$-th dimension on the left-hand side, represented by the function $\text{lhs}_j^k(x)$, must be equal to the corresponding right-hand side dimension $\text{rhs}_j^k$. This ensures consistency between both sides of the equation and results in the following equality constraint:

$$\text{lhs}_j^k(x) = \text{rhs}_j^k(x) \qquad (h)$$

# 3 Solving the Problem

Solving the general integer-valued nonlinear optimization problem is known to be NP-hard (Kannan and Monma 1981). However, for the majority of practical models, the problem structure is predominantly linear and can be efficiently addressed under typical conditions. Moreover, obtaining the global optimum is not strictly necessary. Any method that consistently yields a feasible, finite solution—while maintaining computational complexity independent of the values of resizable parameters—successfully achieves the objective of array-size-independent compilation time. For the results presented in Section 4, a simple greedy hill-climbing algorithm was implemented as a proof of concept, using the original model-defining parameter configuration as the initial starting point.

## 3.1 Utilization of Optimization Results

Once a feasible parameter configuration has been identified—yielding a sufficiently small, array-size-independent model—all affected model sizes must be updated accordingly. In addition, auxiliary structures must be generated and stored to enable recovery of the original array sizes. The bipartite graph used for equation-variable matching, as well as the directed graph used for equation sorting, are both constructed based on these reduced model sizes. This results in a causalization problem that is independent of array sizes. After the causalization is completed, the original array structures are reconstructed using the three-step sorting procedure described in (Abdelhak, Casella, and Bachmann 2023). Notably, the third sorting step differs for resizable equations, as these do not require resolution through the underlying graph. Instead, it is sufficient to analyze the indexed variables they contain and their usage patterns.

**resizable array-equation** If a resizable array equation is solved for the same variable instance in each of its scalar sub-equations, it poses no additional challenge, as the equation body can be resolved using established techniques. However, when explicit solving is not possible, or when different variable instances are involved, the equation should be reformulated in residual form and handled implicitly during simulation.

**resizable for-equation** Let $0 = f(X,Y,I)$ denote the for-equation expressed in residual form, where $\hat{x} \in X$ is the component reference to be solved for, and $X$ is the set of all component references corresponding to the same variable $x$ as $\hat{x}$. The set $Y$ contains all other component references appearing in the equation, and $I$ represents the set of for-equation iterators along with their respective ranges. To determine the execution order of the individual scalar sub-equations within $f$, one examines the elements of $X$ and compares their positions relative to $\hat{x}$, using the indexing information provided by $I$. The component references in $Y$ do not influence the execution order and can therefore be disregarded for this purpose. If multiple variable instances are intended to be solved, the current implementation does not support retaining the equation in resizable form (see Section 3.2). The complete method is presented in Algorithm 1, which determines the execution order for each iterator. The execution order for a given iterator can fall into one of three categories:

a) in the original order (forwards)
b) reverse to the original order (backwards)
c) in any order (arbitrary)

Once determined, the inner equations do not need to be sorted based on the underlying scalar dependency graph, as their execution order is fully defined by the enclosing structure. As shown in Algorithm 1, the algorithm may revert the resizable status of variables used to determine the size of the observed equation. In such cases, this behavior must either be reported to the modeler via an error message—aborting compilation to allow for appropriate model adaptation—or issued as a warning, allowing compilation

**Algorithm 1** Execution Order

Input: component reference to solve for $\hat{x}$
Input: set of component reference of the same variable $X$
Input/Output: set of iterators, their ranges and markings $I$
**if** there is only one component reference $X = \{\hat{x}\}$ **then**
    Mark all iterators in $I$ with **arbitrary**
**else**
    **for** each dimension $d$ in $x$ **do**
        Find the set $O$ of all subscripts in dimension $d$ for all elements of $X$
        Let $I$ be the set of all iterators occuring in $O$
        **if** there is only one iterator $I = \{i\}$ **then**
            compute the set of partial derivatives $D \leftarrow \{\delta o/\delta i | o \in O\}$
            **if** $|D| = 1$ **then**
                compute the set of all offsets $F$ for the elements of $O$
                let $\hat{f} \in F$ be the offset for the occurence of $\hat{x}$
                **if** $\hat{f} \leq f \; \forall f \in F$ and $i$ is not marked **forwards then**
                    Mark $i$ with **backwards**
                **else if** $\hat{f} \geq f \; \forall f \in F$ and $i$ is not marked **backwards then**
                    Mark $i$ with **forwards**
                **else**
                    Mark $i$ with **arbitrary**
            **else**
                The equation is not resizable
                **return**
        **else**
            The equation is not resizable
            **return**

to proceed. In the latter case, the resizable status of the affected variables is reverted, the modified optimization problem is resolved, and the causalization pipeline is subsequently executed again.

**Example 3** *In the trivial case where a for-equation is solved for a variable and no other instances of that variable appear within the equation, the execution order is arbitrary. The following example illustrates a non-trivial case involving two possible scenarios. The equation may be solved either for $x[i]$ or for $x[i+1]$:*

```
for i in 1:p loop
  x[i+1] = x[i]*2;
end for;
```

*If $\hat{x} \equiv x[i+1]$, the structure of the equation body remains unchanged after solving, and the iterator i can be traversed in its original forward order, since each computed element depends only on a previously computed one. Conversely, if $\hat{x} \equiv x[i]$, the iterator range must be traversed in reverse order, as each element otherwise would depend on a value that is computed in a later iteration.*

*Expanding the idea by adding a third indexed instance of x leads to following examplary for-equation:*

```
for i in 1:p loop
  x[i] = x[i+1]*2 + x[i+2]*3;
end for;
```

*If $\hat{x} \equiv x[i+2]$, the execution order is forward, and if $\hat{x} \equiv x[i]$, it is backward—consistent with the behavior described previously. However, if $\hat{x} \equiv x[i+1]$, the computation would depend on both a preceding and a subsequent value, resulting in an algebraic loop. In such cases, since residual equations in an implicit system do not require a specific execution order, the iterator traversal is considered arbitrary.*

## 3.2 Limitations and Future Work

Although this approach is designed to be broadly applicable to general *Modelica* models, certain limitations in its current implementation remain, warranting further investigation and development.

**connect equations** The modeling language *Modelica* allows the construction of complex models through hierarchical composition, using sub-components interconnected via specialized connectors. These connectors implicitly define a set of equations that are incorporated into the model during the flattening process, which transforms the component-based structure into a single mathematical representation suitable for symbolic manipulation and simulation. A


DOI 10.3384/ecp218203

graph-based algorithm has been developed to determine these connection equations. As with other *Modelica* equations, connect-equations can be enclosed within for-equation constructs, potentially depending on resizable parameters. To enable array-size-independent compilation of such models using the optimization approach presented in this paper, the graph-based connection algorithm must be extended to handle undetermined array sizes during compilation.

**split equations** If a multi-dimensional equation must be solved for multiple distinct variable instances, it is partitioned into several slices, with each slice being solved for one of the targeted variables. The current implementation of the optimization does not guarantee that the original equation is of sufficient size to ensure that all resulting slices are fully represented in the minimal version of the model. In principle, additional constraints could be incorporated into the optimization problem to guarantee slice-safety; however, doing so would likely require a comprehensive prior analysis to anticipate all potential slicing operations.

**entwined equations** The concept of entwined equations refers to the evaluation of two or more, potentially multi-dimensional, equations in an alternating sequence (as detailed in (Abdelhak, Casella, and Bachmann 2023)). While, in principle, it is feasible for one or more of these equations to be resizable, the current implementation requires structural enhancements to support the generation of correct and efficient code.

**non-trivial index accessing** The proposed method supports only linear index access, requiring a consistent multiplicative factor for all accesses to solved variables and their occurrences within resizable equations. This constraint is necessary to ensure a clear and strict dependency structure, which is essential for classification into one of the defined execution order categories. While the algorithm could be extended to handle certain fringe cases, such enhancements are generally unnecessary, as the majority of practical *Modelica* models do not demand a more comprehensive approach.

### 3.3 Implementation in OpenModelica

As previously mentioned, the proposed approach has been integrated into the OpenModelica compiler and can be activated using the `--newBackend` compiler flag. Resizable array-defining parameters can be specified individually by applying the annotation
`__OpenModelica_resizable=true`.
Alternatively, all array-defining parameters can be treated as resizable by enabling the `--resizableArrays` compiler flag.

## 4 Results

All experiments were conducted on a laptop with a 12th Gen Intel® Core™ i7-12800H x 20 @ 4.80GHz, 31 GB RAM and a 1TB NVMe SSD, running Ubuntu 20.04.6 LTS. The evaluation was conducted using OpenModelica version 1.24.5-dev, compiled from source.
The `--newBackend` and `--resizableArrays` compiler flags were enabled for the relevant tests.

The presented proof-of-concept implementation was evaluated on several scalable models and consistently produced satisfactory results. The model illustrated in Figure 1, originally developed by Martin Otter, is a simplified variant of the model presented in (Neumayr and Otter 2023). It represents an insulated rod discretized into an array of segments, with the level of discretization controlled by a single parameter, $nT$, which can be designated as resizable. Specifically, $nT$ determines the sizes of the variables $T$ and $Q_{\text{flow}}$, and also defines the bounds for two for-equations, with iterators $i$ in $2 : nT$ and $i$ in $1 : nT$, respectively. As illustrated, Assumption 1 holds, since the resizable parameter $nT$ uniformly influences both variable and equation sizes. The objective function can be derived from Equations 1, 2, and 3, based on the two iterators that define the sizes of the corresponding for-equations. The sizes of all other equations are omitted from the formulation, as they remain constant.

$$\underset{nT}{\arg\min} \quad F(nT) = 2 \cdot nT - 1 \tag{4}$$

The equations and variables impose a variety of constraints on the optimization problem, the majority of which can be omitted due to being trivially satisfied. However, they are presented in the following as illustrative examples of the method. The resizable parameter $nT$ is restricted by a minimal size, therefore a constraint $g_3$ can be formulated:

$$nT \geq 1 \tag{5}$$

which is not the only variable constraint in this model. The equation

```
port_a.Q_flow = 2*k1*(port_a.T-T[1]);
```

does not pose a restriction itself, but the indexed variable $T[1]$ results in a constraint $g_5$:

$$1 \leq nT \tag{6}$$

The same holds for all remaining equations, which involve indexed variables but can be evaluated as trivially satisfied. As such, they do not provide additional illustrative value for the purposes of this example. Each of the two for-equations introduces one constraint, both derived from Equation $g_1$:

$$0 \leq nT - 2 \tag{7}$$

$$0 \leq nT - 1 \tag{8}$$

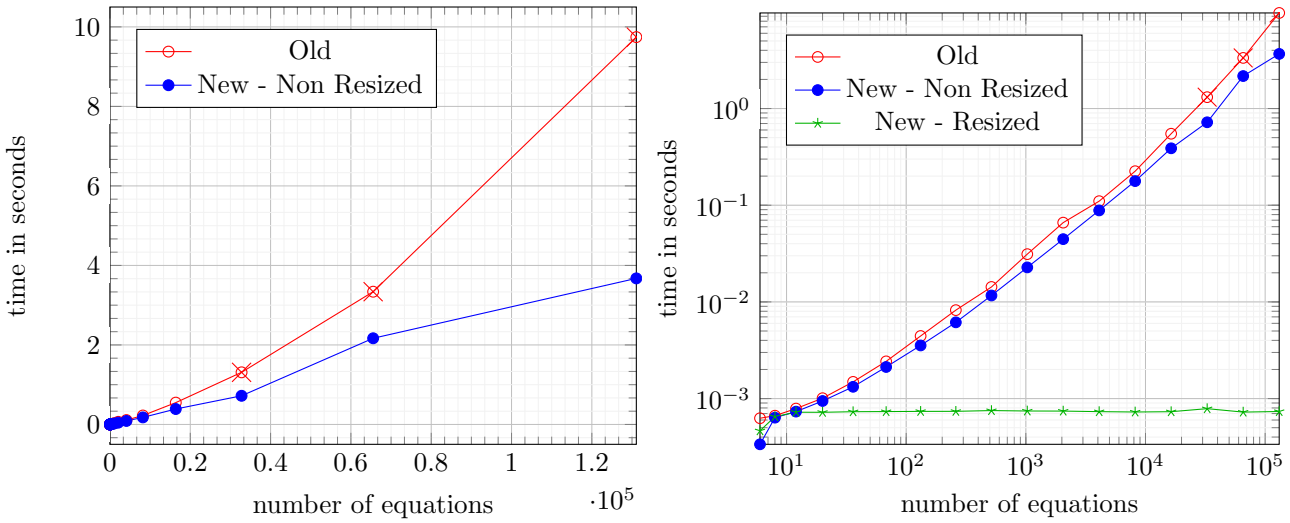
```
model InsulatedRod_equations
"Standard implementaton of an insulated rod with equations"
  import Modelica.Units.SI;
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_a port_a;
  Modelica.Thermal.HeatTransfer.Interfaces.HeatPort_b port_b;
  parameter SI.Length L=1 "lenght of rod";
  parameter SI.Area A=0.0004 "area of rod";
  parameter SI.Density rho=7500 "density of rod material";
  parameter SI.ThermalConductivity lambda=74 "thermal conductivity of material";
  parameter SI.SpecificHeatCapacity c=450 "specifc heat capacity";
  parameter SI.Temperature T0=293.15 "initial temperature";
  parameter Integer nT(min=1)=1000 "number of inner nodes";
  SI.Temperature T[nT](each start=T0, each fixed=true) "temperatures at inner nodes";
  SI.HeatFlowRate Q_flow[nT+1];
protected
  parameter Real dx = L/nT;
  parameter Real k1 = lambda*A/dx;
  parameter Real k2 = rho*c*A*dx;

equation
  port_a.Q_flow = 2*k1*(port_a.T - T[1]);
  port_b.Q_flow = 2*k1*(T[nT] - port_b.T);

  Q_flow[1] =port_a.Q_flow;
  for i in 2:nT loop
    Q_flow[i] = k1*(T[i-1] - T[i]);
  end for;
  Q_flow[nT+1] =-port_b.Q_flow;

  for i in 1:nT loop
    der(T[i]) =(Q_flow[i] - Q_flow[i + 1])/k2;
  end for;
end InsulatedRod_equations;
```

**Figure 1.** Scalable Modelica model representing an insulated rod being heated.




**(a)** Linear Comparison

**(b)** Logarithmic Comparison

**Figure 2.** Comparison of causalization times across three configurations: the old backend, the new backend, and the new backend with support for resizable variables. Nodes marked with a strikethrough indicate that causalization was successfully performed and timed; however, subsequent model building or simulation failed.


DOI 10.3384/ecp218203

Since the model does not contain any array equations, no constraints of the form described in Equation $h$ are present. After omitting all trivially satisfied conditions, the resulting optimization problem can be formulated as follows:

$$\arg\min_{nT} \quad F(nT) = 2 \cdot nT - 1 \tag{9}$$

$$0 \leq nT - 2 \tag{10}$$

The minimal solution to this optimization problem, $nT = 2$, is straightforward to derive and is effectively identified and applied by the *OpenModelica* implementation.

The model presented in Figure 1 has been compiled using different versions of *OpenModelica* to highlight the improvements introduced by the research in this paper. Three configurations are compared: the old backend, the new backend, and the new backend with support for resizable variables. The results of this comparison are presented in Figures 2a and 2b. Figure 2a illustrates the impact of the new backend implementation on causalization time using a linear scale, showing an improvement by a factor of nearly three. Additionally, the updated backend enabled the compilation of larger models, overcoming the scalability limitations of the old backend, which failed to generate code for large models due to memory exhaustion (indicated by 'x' markers in the plots).

While these improvements appear significant, a logarithmic comparison (Figure 2b) reveals that scalability challenges persist: causalization time still increases with array size, despite the reduction of overhead[1]. However, with the introduction of resizable variable support, a fully array-size independent causalization process was achieved, as demonstrated in Figure 2b.

# 5 Conclusions

This work presents an extension to the OpenModelica compiler backend, introducing enhanced support for resizable equations and improving causalization performance for complex array-based models by ensuring that compilation time no longer depends on array sizes. This is accomplished by formulating and solving an integer-based nonlinear optimization problem to determine the minimal feasible model size, which is then treated as resizable prior to simulation, allowing the final model to adapt to arbitrary sizes before runtime.

The method has four key limitations: first, the graph-based connection algorithm needs extension to handle undetermined array sizes during flattening. Second, the optimization may not fully represent all slices when partitioning multi-dimensional equations, requiring additional constraints. Third, entwined equations need structural improvements for efficient code generation. Finally, the method currently supports only linear index access, which is sufficient for most practical *Modelica* models but could be extended for more complex patterns. Despite these, the current implementation is robust enough for most feasible *Modelica* models.

Although the presented algorithm remains in a proof-of-concept stage—with potential improvements in the choice of optimization strategy and possible relaxation of the underlying problem formulation—the results obtained are highly promising. The algorithm has been evaluated on a wide variety of representative Modelica models, including both synthetic benchmarks and simplified real-world industrial examples. In the case of synthetic benchmarks, full resizability was assumed to stress-test the algorithm's capabilities, while for real-world industrial models, only selected array sizes were treated as resizable to reflect practical usage scenarios.

The development of the new OpenModelica backend has led to substantial performance enhancements. Experimental evaluations demonstrate an almost threefold reduction in causalization time, alongside improved scalability that enables the successful compilation of significantly larger models. The principal contribution of this work, however, lies in the introduction of an optimization-based approach for handling resizable variables. This method effectively decouples compilation time from array sizes, yielding additional performance benefits and paving the way for more efficient large-scale model compilation.

In summary, the integration of resizable variable support marks a significant step forward in improving both the robustness and efficiency of the OpenModelica compilation pipeline. Future work may explore more generalized access patterns, improved diagnostic capabilities, and broader support for dynamic model structures.

# Acknowledgements

This work was conducted as part of the OpenSCALING project (Grant No. 01IS23062E) at the University of Applied Sciences and Arts Bielefeld, in collaboration with Linköping University. The authors would like to express their sincere appreciation to both the OpenSCALING project and the Open Source Modelica Consortium (OSMC) for their support, collaboration, and shared commitment to advancing open-source modeling and simulation technologies.

# References

Abdelhak, Karim, Francesco Casella, and Bernhard Bachmann (2023-12). "Pseudo Array Causalization". In: *Modelica Conferences*, pp. 177–188. DOI: 10.3384/ecp204177.

Fritzson, Peter, Adrian Pop, Karim Abdelhak, et al. (2020-10). "The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development". In: *Modeling, Identification and Control: A Norwegian Research Bulletin* 41, pp. 241–295. DOI: 10.4173/mic.2020.4.1.

Kannan, Ravindran and Clyde L. Monma (1981). "On the Computational Complexity of Integer Programming Problems". In: *Optimization and Operations Research*. Vol. 157. Lecture Notes in Economics and Mathematical Systems. Springer,

[1] All other methods in the new backend were already array-size independent, as detailed in (Abdelhak, Casella, and Bachmann 2023)

pp. 161–172. DOI: 10.1007/978-3-642-95322-4_17. URL: https://doi.org/10.1007/978-3-642-95322-4_17.

Marzorati, Denise, Joaquín Fernández, and Ernesto Kofman (2024-06). "Efficient Matching in Large DAE Models". In: *ACM Transactions on Mathematical Software* 50. DOI: 10.1145/3674831.

Modelica Association (2023). *Modelica® Language Specification, Version 3.6*. https://specification.modelica.org/maint/3.6/MLS.pdf. Accessed: 2025-04-18.

Neumayr, Andrea and Martin Otter (2023-12). "Variable Structure System Simulation via Predefined Acausal Components". In: *Modelica Conferences*, pp. 511–520. DOI: 10.3384/ecp204511.