POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Compilation and Optimization of Large Scale Modelica DAE Models

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Author: **Nicola Camillucci**

Student ID: 944569
Advisor: Prof. Giovanni Agosta
Co-advisors: Francesco Casella, Daniele Cattaneo, Stefano Cherubin,
Alberto Leva, Michele Scuttari, Federico Terraneo
Academic Year: 2020-21

Abstract

Large-scale physical phenomena can be modeled by means of differential and algebraic equations systems. Many engineering fields take advantage of modeling to facilitate the prototyping, verification and maintenance of any system, because it allows to know ahead of time the evolution of the behavior of these systems. Due to these needs, a declarative modeling language called Modelica was devised to describe a physical model with a high-level structure and then use this description to produce an accurate simulation. Currently, the Modelica compilers available on the market are unable to keep up with the always increasing complexity of modern-day technologies. This is caused by the inability of such compilers to take advantage of object-oriented structures that compose the model. For this reason, a new compiler has been designed in recent years to overcome these limitations. This compiler, called MARCO, has the main objective of maintaining intact data structures that compose the model as much as possible. Doing so, it is possible to fully exploit the locality properties of modern computer architectures.

MARCO was initially only capable of simulating models through the use of the Forward Euler method. This poses great restrictions on the type of models that can be simulated using this compiler. This document describes how this newly born compiler has been enhanced with an external DAE solver, called IDA, while also maintaining the original objective of preserving the data structures of the physical model. This way, MARCO will be able to create an efficient simulation without posing any restriction on the input models. We will first make an overview of the most used mathematical methods to solve DAE systems and of the state of the art framework technologies for developing compilers. It will then be described how the introduction of an external DAE solver within MARCO was designed, along with some optimizations and implementation details on the most delicate parts. Finally, the correctness and the usefulness of this approach will be demonstrated through experimental tests.

Keywords: Modelica, Compilers, LLVM, MLIR, Modeling, Simulation.

Abstract in Lingua Italiana

I fenomeni fisici su larga scala possono essere modellati mediante sistemi di equazioni differenziali e algebriche. Molti campi dell'ingegneria sfruttano la modellazione per facilitare la prototipazione, la verifica e la manutenzione di qualsiasi sistema, perché questa permette di conoscere in anticipo l'evoluzione del comportamento di questi sistemi. A causa di queste esigenze, è stato ideato un linguaggio dichiarativo di modellazione chiamato Modelica per descrivere un modello fisico con una struttura di alto livello e quindi utilizzare questa descrizione per produrre una simulazione accurata. Al momento, i compilatori Modelica disponibili sul mercato non sono in grado di tenere il passo con la complessità sempre crescente delle tecnologie moderne. Questo è causato dall'impossibilità di tali compilatori di sfruttare le strutture orientate agli oggetti che compongono il modello. Per questo motivo, negli ultimi anni è stato progettato un nuovo compilatore per superare queste limitazioni. Questo compilatore, chiamato MARCO, ha come principale obiettivo il mantenere intatte le strutture dati che compongono il modello il più possibile. In questo modo è possibile sfruttare appieno le proprietà di località delle moderne architetture dei computer.

MARCO inizialmente era solo in grado di simulare modelli attraverso l'uso del metodo di Eulero in avanti. Ciò pone grandi restrizioni sul tipo dei modelli che possono essere simulati tramite questo compilatore. Questo documento descrive come questo nuovo compilatore sia stato migliorato con un risolutore DAE esterno, chiamato IDA, mantenendo l'obiettivo originale di preservare le strutture dati del modello fisico. In questo modo, MARCO sarà in grado di creare una simulazione efficiente senza porre alcuna restrizione sui modelli in ingresso. Per prima cosa faremo una panoramica dei metodi matematici più utilizzati per risolvere i sistemi DAE e delle tecnologie all'avanguardia per lo sviluppo di compilatori. Verrà quindi descritto come è stata progettata l'introduzione di un solutore DAE esterno all'interno di MARCO, insieme ad alcune ottimizzazioni e dettagli implementativi sulle parti più delicate. Infine, la correttezza e l'utilità di questo approccio sarà dimostrata attraverso verifiche sperimentali.

Parole chiave: Modelica, Compilatori, LLVM, MLIR, Modellazione, Simulazione.

Contents

Abstract **i**

Abstract in Lingua Italiana **iii**

Contents **v**

Introduction **1**

1 Background **5**

1.1 ODE Models 5
1.2 DAE Models 8
1.2.1 Algebraic Loops 8
1.2.2 Structural Singularities 10
1.2.3 DAE Solvers 11
1.3 SUNDIALS IDA Library 11
1.4 Modelica Language 13
1.5 Modelica Compilers 17
1.6 Compilers: LLVM and MLIR 19
1.7 MARCO Compiler 21
1.7.1 Front-End: Parsing 22
1.7.2 Middle-End: Model Solving 23
1.7.3 Back-End: Lowering 26

2 Design and Implementation **27**

2.1 Raw DAE Mode 27
2.2 Causalized DAE Mode 28
2.3 Identification of Non-Trivial Variables 30
2.3.1 Matching 30
2.3.2 SCC Resolution 31

2.3.3 Intermediate Passes 32
2.3.4 Scheduling 33
2.4 Trivial Variable Substitution 34
2.5 Sundials IDA Integration 37
2.5.1 MLIR IDA Dialect 39
2.5.2 Residual Function Computation 39
2.5.3 Jacobian Matrix Computation 40
2.5.4 Initialization of IDA Data 43
2.6 MARCO Main Program 46

3 Experimental Results **47**
3.1 Thermal Chip DAE Model 47
3.2 Result Correctness 49
3.3 Binary Size 50
3.4 Compilation Time 53
3.5 Simulation Time 54
3.6 Memory Usage 54

4 Conclusions **59**
4.1 Future Works 60

Bibliography **63**

A ThermalChipDAE Benchmark **69**

List of Figures **73**

List of Tables **75**

List of Listings **77**

List of Algorithms **79**

List of Symbols **81**

Acknowledgments **83**

Introduction

Models and simulations Every natural physical phenomenon that occurs in nature can be described with a set of mathematical equations. These equations can be put together into systems of differential and algebraic equations, called models. In order to observe the evolution of such phenomena we must then simulate the corresponding model for a fixed interval of time. Such simulation can be performed with different methodologies, each of which has different properties such as stability, accuracy and complexity.

The simplest method consists in using the Forward Euler formula, which allows to compute the state of the system at a given time, starting from its state at a previous moment. It is a first order explicit method and can be obtained by approximating derivatives into finite differences. However, this method is only suitable for simulating small models that are highly stable, with weak requirements on accuracy and without any non-explicitable equation. Since we cannot usually make these assumptions, for modeling most of the real-world phenomena, we must employ more advanced techniques [12].

Applications and speed requirements The virtual representation of a physical object or process is often referred to as a "Digital Twin" of the real system [46]. This mechanism, based on simulating the behavior of large and complex systems or machines, even before building them, can be useful in many different engineering fields [13]. For example, in the automotive [47], mechatronic [55], building construction [50] and electrical [10] domains the need of adopting such prototyping mechanisms has grown significantly in recent years. This process does not stop at design time [35], but it can also be extended, for example, to the maintenance [3] and anomaly mitigation [51] of any of these systems.

Hence, in the industrial world arises the need of a standard language and a software framework that is capable of, at the same time, ease the work of a modeler, by providing a series of tools and constructs to describe a complex system in an equation-based form, and simulate efficiently and precisely the evolution of a given system. These needs were met, in the last few decades, by the standardization of a modeling language and by the development of more than one compiler capable of parsing such language and then translating the definition of a model into an executable sequential algorithmic solution [5, 9].

Modelica language Modelica is an equation-based declarative modeling language that allows users to model physical systems in a mathematical fashion through the use of differential and algebraic equations [34, 36]. It also provides some constructs common to imperative languages, such as classes, functions and arrays, that can ease the user's task in the construction of a model. This language allows the modeler to ignore how the system will be translated, solved and simulated, so that they can focus only on modeling the system.

The two main currently available compilers on the market that can parse Modelica and produce an executable simulation are OpenModelica [23] and Dymola [8]. Both these compilers make use of de-vectorization of array of variables and loop-unrolling, which make the compilation of a model non-constant with respect to the size of internal variables. This can be detrimental to the performance of the compiler, especially when dealing with large-scale models with hundreds of thousands of equations [4, 48]. We present in this document an alternative prototype compiler called MARCO (Modelica Advanced Research COmpiler), which does not make use of de-vectorization nor loop unrolling. This allows to keep intact, where possible, arrays and for-loops in order to make use of the efficient corresponding data structures and control flow statements typical of imperative programming languages [19, 49].

DAE systems complexity We can distinguish systems of differential and algebraic equations into Ordinary Differential Equation (ODE) systems and Differential-Algebraic Equation (DAE) systems. The equations contained in the former can always be reordered, explicitated and reduced to a series of assignments. Then, the evolution of the system can be simulated by using a simple integration method such as Forward Euler. With the latter, on the other hand, this procedure cannot generally be performed immediately. Rather, the DAE system needs to either be transformed into an ODE system first, or be simulated as it is with more advanced integration techniques [12].

In particular, to efficiently manage very large and stiff systems, we need methods that are able to adjust their order and time step at runtime, in order to accommodate the stability and accuracy requirements of the considered model. These methods usually allow a more precise execution and fewer iterative steps with respect to methods with constant step size, but they also require a higher computing power. Hence the need of a compiler that contains all the functionalities that the Modelica language offers, along with the generation of an efficient and accurate simulation that does not depend on the size of internal parameters of the system [1].

Proposed solution We will show in this document how to implement such integration mechanisms inside the MARCO compiler, by keeping intact, and taking advantage of, the array data structures characterizing a model. For this purpose, the SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers) [24] package was employed. SUNDIALS is a library containing several solvers, among which IDA, that is able to solve DAE systems using a variable-step-size and variable-order BDF method, which will be used for the scope of this thesis. This method requires the computation of the Jacobian matrix, along with the Residual function of the system.

It will be shown how the IDA solver was integrated inside the MARCO compiler in addition to preserving its original objective, therefore achieving a constant compilation time and binary executable size with respect to the internal parameters of the model. Then, some further optimizations are performed in order to efficiently exploit this integration method. As it will be shown, the temporal complexity of this algorithm strictly depends on the number of variables computed by the IDA solver. Since the majority of physical models only contain a small number of equations that cannot be solved by simple assignments, it is particularly important to reduce this number as much as possible. This can be done by filtering away trivial variables that can be solved without the use of an integration method after the IDA solver took care of the differential and implicit equations.

Document structure This document is organized as follows.

Chapter 1 first refreshes some mathematical concepts that are required to solve and simulate DAE models. Then it presents the standard Modelica language and the currently existing market-ready compilers that make use of this modeling language. Here we also present the LLVM framework that was employed to develop the recently conceived MARCO compiler, explaining how we can improve the simulation performance by taking advantage of vector equations.

In Chapter 2 we present how to extend MARCO with a DAE solver. In particular, we aim at solving index-1 DAE systems with the use of the external library SUNDIALS IDA. For this reason, we will also be required to symbolically compute the Jacobian matrix and the Residual function of the system. We then improve this solution by distinguish trivial and non-trivial equations. This way we can hide from IDA those equations that can be solved with a simple series of assignments, so that we only use the expensive DAE solver to compute the bare minimum number of non-trivial equations.

Chapter 3 presents some results of the implemented compiler when simulating a large and scalable system in order to validate the usefulness of our approach. These results are

also compared with the OpenModelica compiler and a C++ hand-written simulation of the same model, in order to evaluate the performance differences.

Finally, in Chapter 4 we make some final remarks and discuss some future works that will have to be implemented before a first public release of the compiler.

1 | Background

In this chapter we introduce some mathematical concepts and some pre-existing technologies that are useful for pursuing our aim of solving large DAE models in an efficient way by preserving structural information such as arrays and loops. In particular, different ways of solving ODE and DAE systems are shown first. Then we present some existing technologies that tackle the problem of modeling and simulating a physical system. Finally, the MARCO compiler, which will be extended for the purpose of this document, is shown.

1.1. ODE Models

It is possible to model a physical phenomenon by means of differential and algebraic equations. In the most generic form, such a system can be represented as a Differential-Algebraic system of Equations (DAE) that can be written as:

$$\boldsymbol{F}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) = 0 \tag{1.1}$$

where $\boldsymbol{x}$ is the vector of state variables, $\dot{\boldsymbol{x}}$ is the vector of derivatives of the state, $\boldsymbol{v}$ is the vector of algebraic variables, $\boldsymbol{u}$ is the vector of known input variables and t is the time. $\boldsymbol{F}$ is also called the residual function of the system. The two vectors of derivatives and algebraic variables represent the unknowns of the system and can be grouped as $\boldsymbol{y} = [\dot{\boldsymbol{x}}\ \boldsymbol{v}]$, while both the state vector and the input vector are always known at any instant of time and can be regrouped as $\boldsymbol{z} = [\boldsymbol{x}\ \boldsymbol{u}]$. This representation is also called a model.

In some cases, a DAE system can be translated into a simpler form called a Ordinary Differential Equation (ODE) system:

$$\begin{cases} \dot{\boldsymbol{x}}(t) & = \boldsymbol{f}(\boldsymbol{x}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \\ \boldsymbol{v}(t) & = \boldsymbol{g}(\boldsymbol{x}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \\ \boldsymbol{x}(t_0) & = \boldsymbol{x_0} \end{cases} \tag{1.2}$$

where the additional third equation represent the initial conditions of the system at a starting time t_0. By adding the third equation we obtained an Initial Value Problem (IVP) that can be simulated for a given interval of time.

In order to algorithmically compute the evolution of an ODE system, we also need to know how the state variables evolve through the time. This can be done, for example, starting from the derivative operations, by approximating the Taylor series expansion:

$$x(t^* + h) = x(t^*) + \frac{dx(t^*)}{dt} \cdot h + \frac{d^2x(t^*)}{dt^2} \cdot \frac{h^2}{2!} + \cdots + O(h^n) \tag{1.3}$$

and by choosing an arbitrarily small h and a truncation order n. The simplest of these methods is the first-order explicit Forward Euler method, which can be written as:

$$x(t^* + h) \approx x(t^*) + \frac{dx(t^*)}{dt} \cdot h \approx x(t^*) + \dot{x}(t^*) \cdot h \tag{1.4}$$

Let us consider the following IVP of a simple RLC low-pass filter circuit:

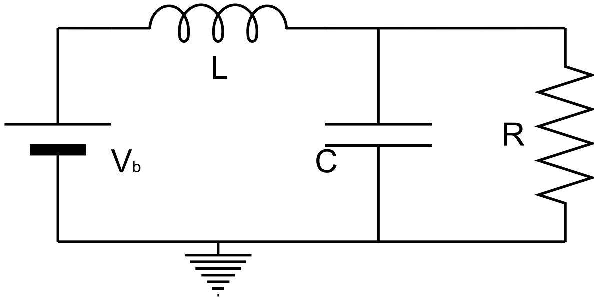


Figure 1.1: Electrical RLC Circuit [54]

$$\begin{cases} eq_1: & V = i_R \cdot R \\ eq_2: & C \cdot \frac{dV}{dt} = i_C \\ eq_3: & L \cdot \frac{di_L}{dt} = (V_b - V) \\ eq_4: & i_L = i_R + i_C \\ ic_1: & i_L(t_0) = i_{L0} \\ ic_2: & V(t_0) = V_0 \end{cases} \tag{1.5}$$

where $\boldsymbol{y} = [\frac{dV}{dt}\, \frac{di_L}{dt}\, i_R\, i_C]$ are the unknown variables of the system and $\boldsymbol{z} = [V\, i_L\, R\, L\, C\, V_b]$ are the known variables. This system can be rewritten in explicit form and then ordered based on the dependencies among the variables of the system. This can be done using the Tarjan algorithm [52]. We we will see more in detail how this algorithm works in a later section. Doing so we obtain:

$$\begin{cases} eq_1: & i_R = V/R \\ eq_4: & i_C = i_L - i_R \\ eq_3: & \dot{i_L} = (V_b - V)/L \\ eq_2: & \dot{V} = i_C/C \end{cases} \tag{1.6}$$

	i_R	i_C	$\dot{i_L}$	$\dot{V}$
eq_1	1	0	0	0
eq_4	1	1	0	0
eq_3	0	0	1	0
eq_2	0	1	0	1

Table 1.1: RLC electrical circuit incidence matrix

The matrix on the right is called Incidence Matrix (IM) of the system, it shows the dependencies among equations and variables. As we can see, all the elements above the main diagonal of matrix 1.1 are zeros. In this particular case the matrix is called Lower Triangular (LT) matrix. Since the incidence matrix of the system is an LT matrix, and there are no implicit equations in the system, we can solve and simulate the model using the Forward Euler method by adding the equations to update the state variables:

$$i_L(t^* + h) = x(t^*) + \dot{i_L}(t^*) \cdot h \tag{1.7}$$

$$V(t^* + h) = x(t^*) + \dot{V}(t^*) \cdot h \tag{1.8}$$

As we can notice, equations 1.6, 1.7 and 1.8 can be easily used as a series of assignments to generate the evolution of the system at any point in time by choosing a start time of the simulation t_0, an end time t_n and a time step h.

Unfortunately, the Euler method is not suitable to solve every model. In particular, it is not able to solve systems containing non-explicitable equations and systems containing algebraic loops, that is, systems whose incidence matrix cannot be transformed into an LT matrix. Furthermore, the stability of the system and accuracy requirements may force the use of a very short time step, increasing the number of iterations that must be executed and thus increasing the overall simulation execution time. This is often the case when dealing with stiff systems [27].

Other single-step integration methods exist that address some of these problems when solving ODE systems. Notably, the Backward Euler formula is a first-order integration method that has a greater stability domain, and the Runge-Kutta algorithm [18] is a predictor-corrector method that is capable of higher accuracy. We will not show in details such methods since they are outside the scope of this thesis, but it could be interesting to consider and evaluate their behavior as a continuation of this thesis. There also exist multi-step methods that use multiple previous states of the system in order to compute the next value of the state vector based on the differential equations. The most commonly used one is the Backward Difference Formulae (BDF). Finally, implicit equations can be solved with the use of nonlinear solvers such as the Newton's method, which is an iterative procedure that finds the roots of a function through successive approximation

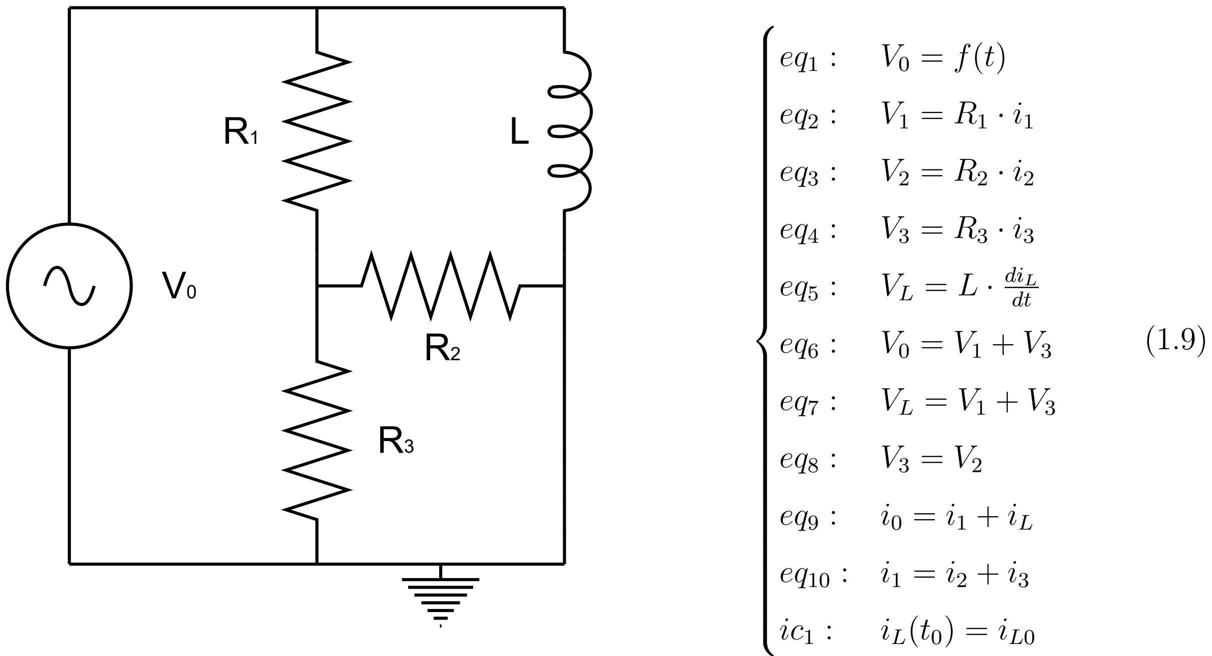
starting from an initial guess. Both the BDF method and the Newton iteration will be used through the SUNDIALS library, which will be introduced in section 1.3, to solve DAE models.

1.2. DAE Models

As mentioned in the previous section, not all the DAE systems can be easily translated into explicit ODE system. In particular, systems containing algebraic loops or structural singularities are the ones that pose the major difficulties in the translation. Furthermore, even if possible, we may want anyway to not perform such translation and solve DAE models as they are, because the translation process could be particularly difficult, or it could increase the computational complexity of the simulation.

1.2.1. Algebraic Loops

The first problem is to efficiently solve algebraic loops. Consider the following electrical circuit with three resistances and one inductor:



$$
\begin{cases}
eq_1: & V_0 = f(t) \\
eq_2: & V_1 = R_1 \cdot i_1 \\
eq_3: & V_2 = R_2 \cdot i_2 \\
eq_4: & V_3 = R_3 \cdot i_3 \\
eq_5: & V_L = L \cdot \frac{di_L}{dt} \\
eq_6: & V_0 = V_1 + V_3 \\
eq_7: & V_L = V_1 + V_3 \\
eq_8: & V_3 = V_2 \\
eq_9: & i_0 = i_1 + i_L \\
eq_{10}: & i_1 = i_2 + i_3 \\
ic_1: & i_L(t_0) = i_{L0}
\end{cases} \tag{1.9}
$$

Figure 1.2: Electrical RL Circuit [12]

where $\boldsymbol{y} = [V_0\, V_1\, i_1\, V_2\, i_2\, V_3\, i_3\, V_L\, \frac{di_L}{dt}\, i_0]$ are the unknown variables of the system and $\boldsymbol{z} = [R_1\, R_2\, R_3\, L\, i_L]$ are the known variables. As we did in the previous section, after explicitating and reordering the model, where possible, the system of equation and the incidence matrix appear as follows:

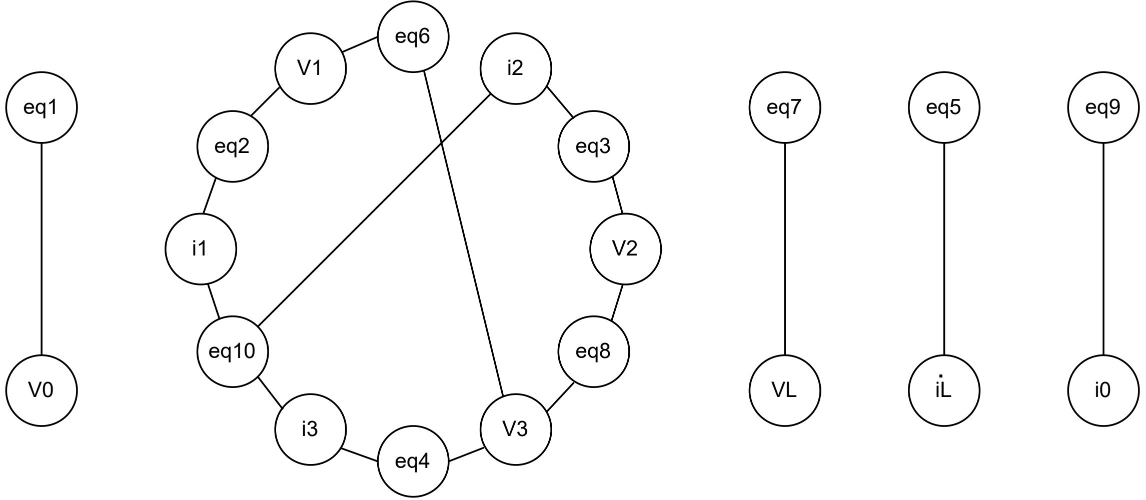


Figure 1.3: SCCs found by the Tarjan algorithm in the RL electrical circuit

$$\begin{cases} eq_1: & V_0 = f(t) \\ eq_2: & V_1 - R_1 \cdot i_1 = 0 \\ eq_3: & V_2 - R_2 \cdot i_2 = 0 \\ eq_4: & V_3 - R_3 \cdot i_3 = 0 \\ eq_6: & V_1 + V_3 = V_0 \\ eq_8: & V_2 - V_3 = 0 \\ eq_{10}: & i_1 - i_2 - i_3 = 0 \\ eq_7: & V_L = V_1 + V_3 \\ eq_5: & V_L = L \cdot \frac{di_L}{dt} \\ eq_9: & i_0 = i_1 + i_L \end{cases} \quad (1.10)$$

	V_0	V_1	i_1	V_2	i_2	V_3	i_3	V_L	$\dot{i}_L$	i_0
eq_1	1	0	0	0	0	0	0	0	0	0
eq_2	0	1	1	0	0	0	0	0	0	0
eq_3	0	0	0	1	1	0	0	0	0	0
eq_4	0	0	0	0	0	1	1	0	0	0
eq_6	1	1	0	0	0	1	0	0	0	0
eq_8	0	0	0	1	0	1	0	0	0	0
eq_{10}	0	0	1	0	1	0	1	0	0	0
eq_7	0	1	0	1	0	0	0	1	0	0
eq_5	0	0	0	0	1	0	0	1	1	0
eq_9	0	0	1	0	0	0	0	0	0	1

Table 1.2: RL electrical circuit incidence matrix

As we can see from the incidence matrix, the Tarjan algorithm [52] is not able to explicitate and schedule 6 of the 10 variables and equations. This is because the system forms a Block Lower Triangular (BLT) matrix instead of an LT matrix. A BLT matrix is an $n \times n$ incidence matrix with only zeros above the diagonal composed of many $m_i \times m_i$ blocks. This is caused by the presence of an algebraic loop inside the system. An algebraic loop is also known as Strongly Connected Component (SCC) in graph theory, and can be represented in an undirected graph as shown in figure 1.3.

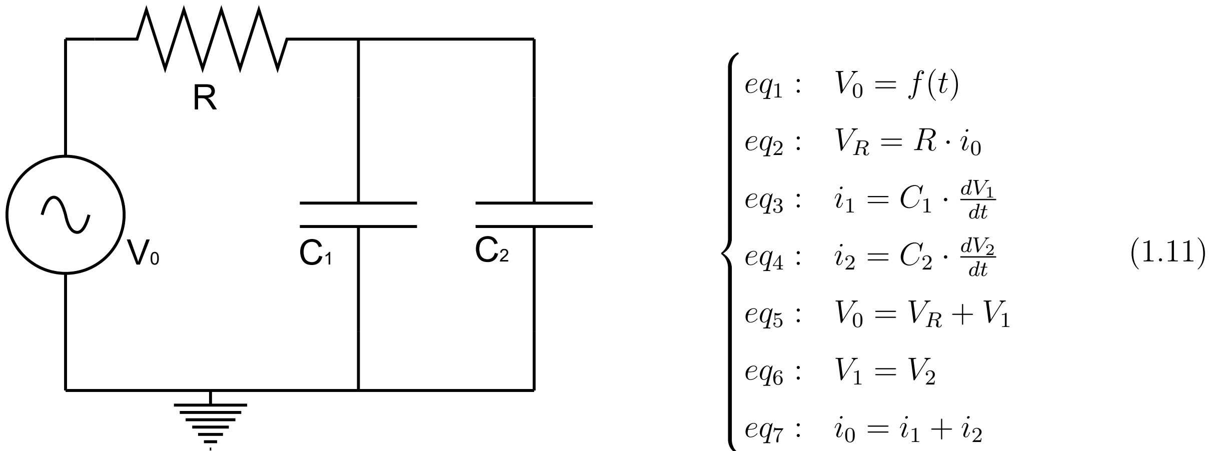
This problem can be solved with two different methods. The first one is the Tearing algorithm [26]. It consists in choosing one or more equations and consider them solvable

for one of their variables, called tearing variable. This way, the remaining equations no longer form an algebraic loop and the Tarjan algorithm can finish reordering them. Finally, the equations are substituted into each other, starting from the one chosen at the beginning, until it is no longer dependent from the other variables of the SCC. The second is the Relaxation algorithm [42], that makes use of the Gaussian elimination to directly solve the system of equations. However, the former method may make equations grow too much in size, while the latter can only solve linear systems. Furthermore, the presence of algebraic loops is very common in electrical systems and these can often reach frightening dimensions in model such as mechanical multibody systems [40, 41].

1.2.2. Structural Singularities

Structural singularities arise when a DAE system contains constraint equations, that is, we are not completely free to choose the initial value of some of the state variables, but we must respect some given constraints. This is especially common in electrical and mechanical systems.

Consider another example of a circuit with one resistance and two capacitors:



$$\begin{cases} eq_1: & V_0 = f(t) \\ eq_2: & V_R = R \cdot i_0 \\ eq_3: & i_1 = C_1 \cdot \frac{dV_1}{dt} \\ eq_4: & i_2 = C_2 \cdot \frac{dV_2}{dt} \\ eq_5: & V_0 = V_R + V_1 \\ eq_6: & V_1 = V_2 \\ eq_7: & i_0 = i_1 + i_2 \end{cases} \tag{1.11}$$

Figure 1.4: Electrical RC Circuit [12]

where $\boldsymbol{y} = [V_0\, V_R\, i_0\, i_1\, i_2\, \frac{dV_1}{dt}\, \frac{dV_2}{dt}]$ are the unknown variables of the system and $\boldsymbol{z} = [V_1\, V_2\, R\; C_1\, C_2]$ are the known variables. As we can see, in eq_6 the two state variables V_1 and V_2 are not free but constrained by this equation. Furthermore, if we do not consider eq_6, which does not determine any unknown variable, we are left with a system with 6 equations and 7 variables. We need therefore to extract a useful equation from the constraint equation.

In mathematical literature, the perturbation index measures the number of constraints among equations. In particular, ODE systems as seen in section 1.1 are called index-0

DAE models, systems containing algebraic loops as seen in section 1.2.1 are called index-1 DAE models and systems also containing structural singularities as seen in this section are called higher-index DAE models [6].

It is possible to reduce the index of higher-index DAE models with the use of the Pantelides algorithm [43]. This method consists in differentiating the constraint equation and adding the newly obtained equation to the system. Then, since an extra equation has been introduced, one of the newly introduced derivatives must be transformed into a Dummy Derivative [33]. Thus, the original constraint equation becomes a normal equation with at least one unknown variable. This process may need to be repeated more than once for DAE systems with index greater than two, until a model without structural singularities is reached. As a side effect, this algorithm often introduces new algebraic loops inside the system.

With the use of such algorithm, we are always able to reduce higher-index DAE models down to index-1 DAE models. For this reason, for the rest of this document, we will only consider index-1 DAE systems.

1.2.3. DAE Solvers

For the purpose of simulating a model, instead of transforming its index-1 DAE system into an ODE system, it may be better to solve the DAE model directly. This is especially true if the model is particularly large or stiff. If we want to do so, we need an implicit algorithm that is able to handle differential equation, algebraic implicit nonlinear equation and algebraic loops altogether.

One of the most famous simulation codes for DAE systems is DASSL (Differential Algebraic System SoLver) [44]. DASSL is a simulation algorithm that implements the BDF formulae of orders 1 through 5 in their DAE form. It is a variable-order and variable-step-size code that uses order buildup during the startup phase. This is exactly what we were looking for. For the purpose of this thesis, the SUNDIALS suite will be used, which contains several systems of equation solvers, among which a reimplementation in C of a variant of the DASSL algorithm, called IDA.

1.3. SUNDIALS IDA Library

SUNDIALS is a SUite of Nonlinear and DIfferential/ALgebraic equation Solvers [24, 30]. It contains a total of six solver that solve different types of system of equations similar to those described in the previous sections. In particular:

Listing 1.1 C prototype of the user-defined IDA Residual function

```
int residualFunction(
        realtype tt, // Current time value.
        N_Vector yy, // Input variables vector.
        N_Vector yp, // Input derivatives vector.
        N_Vector rr, // Output Residual functions vector.
        void* userData // Opaque pointer to user defined data.
);
```

- **CVODE** solves initial value problems for ODE systems.
- **CVODES** is a superset of CVODE that contains sensitivity analysis capabilities.
- **ARKODE** solves initial value ODE problems with additive Runge-Kutta methods.
- **IDA** solves initial value problems for DAE systems.
- **IDAS** is a superset of IDA that contains sensitivity analysis capabilities.
- **KINSOL** solves nonlinear algebraic systems.

The IDA (Implicit Differential-Algebraic) solver [25] is a general purpose library for IVP problems of DAE system presented in the form of equation 1.1. It is implemented in C and its algorithm is based on the DASPK solver [7], which in turn derives from DASSL. The integration method used in IDA is the variable-order, variable-coefficient BDF in fixed-leading-coefficient form [6], where the method order varies between 1 and 5. It makes use of heuristics and supports sparse algebra through the use of the sparse linear algorithm KLU [15].

This library allows the user to select which linear and nonlinear solver to use, the default being the Modified Newton iteration. The user must also provide two callback functions that tell the solver how the system can be computed. The first function must compute the Residual function vector of the system, with the signature shown in listing 1.1.

The second callback function must compute the Jacobian matrix of the system. The Jacobian matrix of the generic DAE system shown in equation 1.1 is a $n \times n$ matrix obtained as follows:

$$\boldsymbol{J} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} = \begin{bmatrix} \frac{\partial \boldsymbol{F}}{\partial y_1} & \dots & \frac{\partial \boldsymbol{F}}{\partial y_n} \end{bmatrix} = \begin{bmatrix} \nabla^T F_1 \\ \vdots \\ \nabla^T F_n \end{bmatrix} = \begin{bmatrix} \frac{\partial F_1}{\partial y_1} & \dots & \frac{\partial F_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial y_1} & \dots & \frac{\partial F_n}{\partial y_n} \end{bmatrix} \tag{1.12}$$

In particular, the Jacobian matrix that is needed by IDA follows this approximation:

Listing 1.2 C prototype of the user-defined approximated IDA Jacobian matrix

```
int jacobianMatrix(
        realtype tt, // Current time value.
        realtype cj, // Alpha parameter.
        N_Vector yy, // Variables vector.
        N_Vector yp, // Derivatives vector.
        N_Vector rr, // Input Residual function vector.
        SUNMatrix JJ, // Output sparse Jacobian matrix.
        void* userData, // Opaque pointer to user defined data.
        N_Vector tempv1, // Temporary auxiliary vector.
        N_Vector tempv2, // Temporary auxiliary vector.
        N_Vector tempv3 // Temporary auxiliary vector.
);
```

$$\boldsymbol{J} = \frac{\partial \boldsymbol{G}}{\partial \boldsymbol{y}} = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} + \alpha \frac{\partial \boldsymbol{F}}{\partial \dot{\boldsymbol{y}}} \tag{1.13}$$

where α is an internal parameter of IDA that depends on the step size and on the method order. Listing 1.2 shows the signature of the callback function that compute the approximated Jacobian matrix required by IDA.

This solver also requires the user to set some simulation parameters. In particular the user must specify the problem dimension, that is the number of equations NEQ and the number of non-zero values inside the Jacobian matrix NNZ, the start time t_0 and the end time t_n of the simulation, the relative and absolute tolerance of the simulation, which influence the step size and the accuracy of the solver. The user must also specify the initial values of vector $\boldsymbol{y}$. For algebraic variables, IDA contains a routine that is able to compute consistent initial condition, therefore only initial guesses of such variables are needed.

It is also possible to specify some optional simulation parameters if we want to guide the simulation, such as the initial step size, the maximum order of the BDF, the maximum number of steps taken and the maximum number of errors allowed. Finally, the user can also query the solver after the simulation to gather information such as the number of steps taken, the step sizes across the simulation, the number of nonlinear iterations and the number of Residual and Jacobian evaluations.

1.4. Modelica Language

Modelica [36] is a declarative, object-oriented, multi-domain modeling language, developed for component-oriented modeling of complex systems, both for academic and indus-

Listing 1.3 RLC circuit model [54]

```
model RLC "A resistor-inductor-capacitor circuit model"
  type Voltage = Real(unit = "V");
  type Current = Real(unit = "A");
  type Resistance = Real(unit = "Ohm");
  type Capacitance = Real(unit = "F");
  type Inductance = Real(unit = "H");

  parameter Voltage Vb = 24 "Battery voltage";
  parameter Resistance R = 100;
  parameter Inductance L = 1;
  parameter Capacitance C = 1e-3;

  Voltage V(start = 0, fixed = true);
  Current i_L(start = 0, fixed = true);
  Current i_R;
  Current i_C;

equation
  V = i_R * R;
  C * der(V) = i_C;
  L * der(i_L) = (Vb - V);
  i_L = i_R + i_C;

annotation(
  experiment(StartTime = 0, StopTime = 1, Tolerance = 1e-6, Interval = 0.001));
end RLC;
```

trial usage. This language was first designed by Hilding Elmqvist in 1996 [34] and it is currently maintained by the Modelica Association [37]. It allows users to model physical systems using a set of parameters, variables and differential and algebraic equations. For example, the electrical RLC circuit of equation 1.5 can be written as shown in listing 1.3. Here we can see the user defined types, parameters, variables and equations composing the model. The *annotation* keyword allows the modelers to also specify how the model will be simulated. Note that, differently from imperative languages, the equal sign $=$ does not represent an assignment operation nor it states causality among variables. It is just the declaration of an equation.

Modelica also allows users to declare parametric multidimensional arrays, for example to discretize the length of a rod as shown in listing 1.4. This allows to declare several scalar variables at once. We can also perform vector operation using such arrays, and also iterate through the arrays with the *for* statement. Several ways of doing so are shown in listing 1.4. This allows modelers to write systems which size can be easily controlled by changing a small set of parameters without changing its physical behavior. This is particularly important for testing the scalability of the system. Differently from imperative languages, for-loops in Modelica do not represent control flow inside the simulation, they are simply a way to declare multiple equations with fewer statements and in a more readable way.

Listing 1.4 Heat transfer in a one-dimensional rod [54]

```
model RodForLoop "Modeling heat conduction in a rod using a for loop"

  // Type and parameter declarations [...]

  parameter Integer n = 10;
  Heat Qleft[n];
  Heat Qright[n];
  Heat Qconv[n];
  Temperature T[n];

initial equation
  T = linspace(200,300,n);

equation
  Qleft[1] = 0;
  for i in 2:n loop
     Qleft[i] = -k * A_c * (T[i] - T[i-1]) / (L / n);
  end for;

  Qright[1:end-1] = -k * A_c * (T[1:end-1] - T[2:end]) / (L / n);
  Qright[end] = 0;

  Qconv = {-h * A_s * (T[i] - Tamb) for i in 1:n};

  rho * V * C * der(T) = Qconv + Qleft + Qright;
end RodForLoop;
```

Models can also be written as reusable parametric classes in an object-oriented fashion. In this way we can simulate multiple instances of the same class, and it is possible to connect different instances among each other with the *connect* operator. As shown in listing 1.5, the same model circuit as 1.3 can be rewritten by modeling the resistor, inductor and capacitor components and connecting them. This is particularly useful for modeling complex system with multiple machineries, so that each independent component can be modeled, simulated and tested separately. Several general-purpose and well-known models are already implemented inside the Modelica Standard Library [22], so that a modeler can simply import them, configure some parameters and connect them to other components. As in other common imperative languages, classes can also be extended, but the semantics of this operation are unusual. For example in Modelica, when extending a class, the fields or attributes of the parent class may also change. Even single instances of a class may also be extended locally. Therefore, in general there are no assumed data layout commonalities between instances of the same class.

Finally, Modelica also allows to write functions that work in an algorithmic way. These functions can take as input both scalars and arrays, even of variable sizes. This functionality is particularly useful in physical systems where some variables are computed by an iterative procedure. This behavior is common, for example, when computing fluid properties [11]. An example algorithmic function implementing a generalized polynomial

Listing 1.5 Object-oriented RLC circuit model [54]

```
package RLCPackage

  // Type declarations [...]

  connector Pin "Pin of an electric component"
    Voltage v "Potential at the pin";
    flow Current i "Current flowing into the pin";
  end Pin;

  partial model OnePort "Partial generic electrical component"
    Voltage v "Voltage drop of the two pins";
    Current i "Current flowing from pin p to pin n";
    Pin p "Positive electrical pin";
    Pin n "Negative electrical pin";
  equation
    v = p.v - n.v;
    i = p.i;
    p.i + n.i = 0 "Conservation of charge";
  end OnePort;

  model Inductor "An inductor model"
    extends OnePort;
    parameter Inductance L;
  equation
    L * der(i) = v;
  end Inductor;

  // Resistor, Capacitor, ConstantVoltage, Ground declarations [...]

  model RLC "A resistor-inductor-capacitor circuit model"
    ConstantVoltage source(Vb = 24);
    Resistor resistor(R = 100);
    Inductor inductor(L = 1, i(start = 0, fixed = true));
    Capacitor capacitor(C = 1e-3, v(start = 0, fixed = true));
    Ground ground;
  equation
    connect(inductor.n, resistor.n);
    connect(capacitor.n, inductor.n);
    connect(inductor.p, source.p);
    connect(capacitor.p, ground.ground);
    connect(resistor.p, ground.ground);
    connect(source.n, ground.ground);
  end RLC;
end RLCPackage;
```

Listing 1.6 Algorithmic computation of a polynomial [54]

```
function Polynomial "Create a generic polynomial from coefficients"
  input Real x "Independent variable";
  input Real c[:] "Polynomial coefficients";
  output Real y "Computed polynomial value";
protected
  Integer n = size(c, 1);
algorithm
  y := c[1];
  for i in 2:n loop
    y := y * x + c[i];
  end for;
end Polynomial;
```

equation is shown in listing 1.6.

1.5. Modelica Compilers

All Modelica compilers currently available on the market share the same structure. They all have a front-end that is able to parse a Modelica file and transform all syntactic sugar that the language offers into a simplified model, called flattened model. Then the middle-end solves the flattened model using the desired method. Finally, the solved model is transformed into executable code implementing the simulation by the back-end. Let us look at this compilation pipeline in greater detail:

- **Parsing:** First, the Modelica model is parsed and transformed into an Abstract Syntax Tree (AST). Here some type checking is also performed to verify the correctness of the equations inside the model.
- **Flattening:** In this phase, all object-oriented structures are simplified. All array variables are transformed into scalar variables (de-vectorization) and all for-loops are transformed into lists of scalar equations (loop unrolling). For example the model described in listing 1.4 will be transformed into something like listing 1.7. This greatly deteriorates the performance of the generated code since, especially when dealing with large models, hundreds of thousands of lines of code are generated, most of which are duplicated code [4].
- **Matching:** This step computes which scalar variable is determined by which scalar equation. Several matching algorithms exists [21], most of which consist in considering the problem as a bipartite graph matching problem.
- **SCC resolution:** Here algebraic loops are found and solved, for example with the Tearing algorithm described in section 1.2.1.
- **Scheduling:** In this phase, every scalar equation is explicitated and ordered accordingly to their mutual dependencies, so that the system can be sequentially simulated.

Listing 1.7 Heat transfer in a one-dimensional rod, flattened model

```
class RodForLoop "Modeling heat conduction in a rod using a for loop"

  // Scalar paramter declarations [...]

  final parameter Integer n = 10;
  Real Qleft[1](unit = "J");
  Real Qleft[2](unit = "J");
  Real Qleft[3](unit = "J");
  Real Qleft[4](unit = "J");
  Real Qleft[5](unit = "J");

  // [...]

  rho * V * C * der(T[6]) = Qconv[6] + Qleft[6] + Qright[6];
  rho * V * C * der(T[7]) = Qconv[7] + Qleft[7] + Qright[7];
  rho * V * C * der(T[8]) = Qconv[8] + Qleft[8] + Qright[8];
  rho * V * C * der(T[9]) = Qconv[9] + Qleft[9] + Qright[9];
  rho * V * C * der(T[10]) = Qconv[10] + Qleft[10] + Qright[10];
end RodForLoop;
```

This is done through the use of the Tarjan algorithm described in section 1.1.

- **Lowering:** Finally, the executable code that simulates the model using the specified method and parameters is generated.

The two most used modeling and simulation environment for the Modelica language are OpenModelica and Dymola. OpenModelica [23, 39] is a free and open source software package developed by the Open Source Modelica Consortium. Among various tools, it contains the OpenModelica Compiler (OMC), a Modelica compiler capable of generating C code for the simulation of a model starting from a Modelica source file. It is written in MetaModelica [45], a superset of the Modelica language that contains additional features that are deemed useful or necessary for writing a compiler, such as exception handling. For this reason, OMC can be considered a bootstrapping compiler. Dymola [8, 14], on the other hand, is a proprietary software product currently maintained by the French company Dassault Systèmes. It offers similar features to those offered by OpenModelica. Other lesser known Modelica compilers also exist such as JModelica [56], currently closed source and maintained by Modelon AB, and Modia [17], open source and written in the Julia programming language. All these four compilers make use of de-vectorization and loop unrolling during the flattening stage. A market-ready compiler capable of keeping intact arrays and for-loops, in order to maintain a constant compilation time with respect to the sizes of the arrays, and to exploit the locality properties of the generated simulation code, has not been introduced yet.

Listing 1.8 LLVM IR of the recursive factorial function

```
define i32 @factorial(i32 %n) {
entry:
  %retval = alloca i32, align 4
  %cmp = icmp sle i32 %n, 1
  br i1 %cmp, label %if.then, label %if.end

if.then:
  store i32 1, i32* %retval, align 4
  br label %return

if.end:
  %sub = sub i32 %n, 1
  %call = call i32 @factorial(i32 %sub)
  %mul = mul i32 %n, %call
  store i32 %mul, i32* %retval, align 4
  br label %return

return:
  %tmp = load i32, i32* %retval, align 4
  ret i32 %tmp
}
```

1.6. Compilers: LLVM and MLIR

A compiler implements a transformation from a high-level source program to a low-level target program. LLVM is a collection of modular and reusable compiler and toolchain technologies written in C++ that helps in implementing such transformation [28, 31]. It is designed around a language-independent Intermediate Representation (LLVM-IR) that serves as a portable, high-level assembly language. This IR is strongly typed and Static Single Assignment (SSA) based. The LLVM-IR was designed for supporting lightweight runtime optimizations, cross-function and interprocedural optimizations, whole program analysis, and aggressive restructuring transformations. A big advantage of this intermediate representation is that an optimizer is not constrained by either a specific source language or a specific target machine. A simplified example of the LLVM-IR implementing a recursive factorial function is shown in listing 1.8.

LLVM can be used to develop a front-end for any programming language, where a source file is translated first into an Abstract Syntax Tree (AST), then into the aforementioned IR. During this phase, the compiler also verifies the correctness of the syntax and the semantic of the source language, eventually reporting any errors found. In the middle-end, optionally, such IR can be optimized with a variety of transformations over multiple passes. These transformations operate from LLVM-IR to LLVM-IR. Several optimization

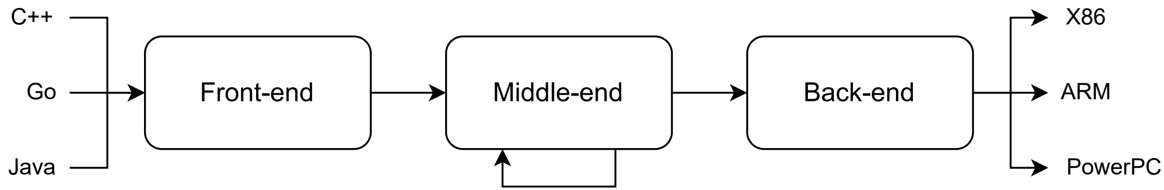


Figure 1.5: Three stage LLVM-based compiler pipeline

passes are already available in the LLVM project, such as dead code elimination and constant propagation, and it is then also possible to implement and use custom made ones. Finally, LLVM also provides a back-end for every instruction set architecture, where further target-dependent optimizations can be performed, and then the optimized IR is translated into assembly code. A representation of this three stage pipeline is shown in figure 1.5.

Among the various sub-projects of LLVM, MLIR is of particular relevance when it comes to writing compilers. MLIR (Multi-Level IR) is a novel approach to building reusable and extensible compiler infrastructures [29, 32]. It aims at facilitating the design and implementation of code generators, translators and optimizers at different levels of abstraction and across application domains, hardware targets and execution environments. This is done by giving the possibility of writing a custom intermediate representation called dialect. A dialect is a language made of operations, types, attributes and traits, each of which can have a custom semantic. This way, it is possible to write a higher-level IR with all the characteristics needed by a specific application. It is then possible to write optimization passes within the dialect, or from a dialect to another, through pattern conversion, and only at end translate such dialects into the LLVM-IR. Listing 1.9 shows an example of the factorial function 1.8 written with a toy MLIR dialect.

MLIR already provides several useful dialects. Most notably, the “built-in” dialect is automatically included in any program and it contains the core types, attributes and operations of an MLIR program, the “scf” dialect contains the operations for structured control flow, and the “llvm” dialect directly maps all types and operations available in LLVM-IR. MLIR also provides the passes to convert such dialects first to the “llvm” dialect, then to LLVM-IR. Therefore, a user is only required to write passes that translate a custom dialect to any dialect contained in the MLIR repository and then call such standard conversion passes in order to generate the final executable file.

Listing 1.9 MLIR IR of the factorial function with a toy dialect

```
toy.function @factorial(%arg0 : !toy.int) -> (!toy.int) {
  %0 = toy.member_create : !toy.member<stack, !toy.int>
  %1 = toy.constant #toy.int<1>
  %2 = toy.lte %arg0, %1 : (!toy.int, !toy.int) -> !toy.bool
  toy.if (%2 : !toy.bool) {
    %3 = toy.constant #toy.int<1>
    toy.member_store %0, %3 : !toy.member<stack, !toy.int>
    toy.yield
  } else {
    %4 = toy.constant #toy.int<-1>
    %5 = toy.add %arg0, %4 : (!toy.int, !toy.int) -> !toy.int
    %6 = toy.call @factorial(%5) : !toy.int -> !toy.int
    %7 = toy.mul %arg0, %6 : (!toy.int, !toy.int) -> !toy.int
    toy.member_store %0, %7 : !toy.member<stack, !toy.int>
    toy.yield
  }
  %8 = toy.member_load %0 : !toy.member<stack, !toy.int> -> !toy.int
  toy.return %8 : !toy.int
}
```

1.7. MARCO Compiler

As anticipated in section 1.5, all the currently available Modelica compilers make use of de-vectorization and loop unrolling when solving a given model. This allows to easily solve an ODE or DAE system, but it drastically increases the compilation time and the binary size of the simulation executable. Furthermore, this makes it impossible to exploit the temporal and spatial locality of the cache memory of modern processors, thus also slowing the simulation [16].

To address this problem, in the past years our research group (the PoliMi Modelica Working Group) has introduced a new Modelica compiler called MARCO (Modelica Advanced Research COmpiler), written from scratch, with performance and scalability as its main goals [19, 49]. This was done because currently available open-source compilers, such as OMC, are too complex to be easily adapted to our needs, since they also include many features that are orthogonal to the objective of our work.

The MARCO compiler is based on the LLVM compiler framework and also makes use of MLIR with a custom dialect, called Modelica dialect, as an intermediate representation of the internal state during the optimization passes. The main feature of this compiler is that, differently from the others, it does not make use of de-vectorization and loop unrolling during the flattening phase. This allows to retain structural information of the original Modelica source file, such as arrays and for-loops, thus allowing the generation

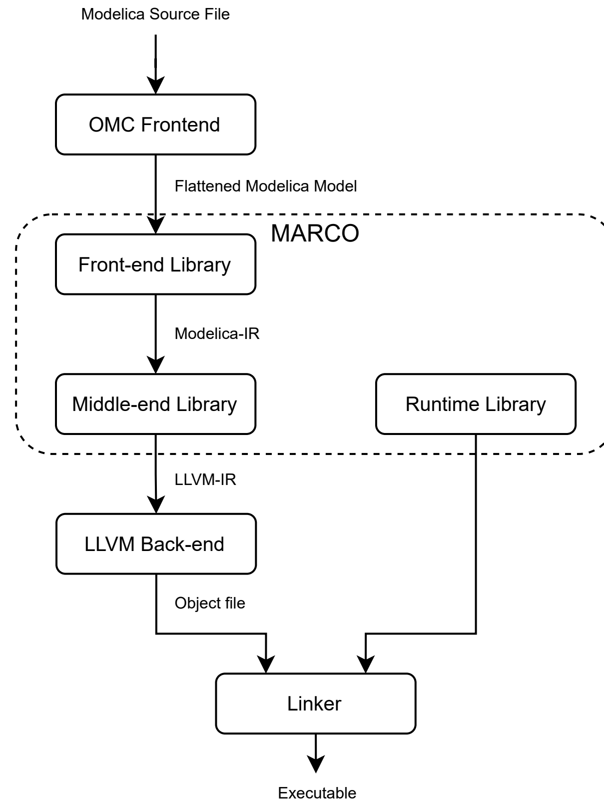


Figure 1.6: MARCO pipeline

of machine code that is optimized for the target hardware architecture and allowing a constant compilation time.

MARCO is, for now, only able to parse a subset of the Modelica language, which includes models, equations, variables and, more recently, also functions. This are the essential elements to write a complete model and perform a full simulation of a system. Other elements such as records, which are particularly useful for modeling a system in an object-oriented fashion, will be implemented inside MARCO in the near future. The compilation pipeline of MARCO, similar to the one of a generic compiler, can be partitioned in three phases: front-end, middle-end and back-end. A diagram of this pipeline is shown in figure 1.6.

1.7.1. Front-End: Parsing

Before entering the front-end of the MARCO compiler, we first take advantage of the flattening stage of the OMC compiler to remove unsupported features, such as object-oriented code and classes. In particular, we make use of the flag *-d=nonfScalarize*, which disables the scalarization pass during the flattening process. This allows to retain array

structures and for-loops inside the models.

The next step is to build an AST starting from the flattened source Modelica file. During this phase, a type checking pass is also performed in order to verify the semantic correctness of the given model. Some preliminary transformations are also applied on the modified AST such as constant folding and constant propagation.

Then, the intermediate representation of the model is built starting from the AST. This phase is called code generation pass. The generated IR, called Modelica-IR, is based on a MLIR Modelica dialect built specifically for this project [49], and describes the data structures contained in the original model – namely the variables, equations, parameters and functions – at a high level of abstraction. In this way we can represent all constructs of Modelica that we intend to support with a data structure that can be easily extended in the future. This high-level IR of the system also allows to perform optimization passes hiding implementation details, before converting it to lower-level representations.

1.7.2. Middle-End: Model Solving

The middle-end of MARCO is where several optimization and transformation passes are performed. We first introduce some definitions that will be of great use also in the next chapter.

Definition 1.1 (Rank). *The rank R of a multidimensional array or a nested for-loop is, respectively, the number of dimensions of the array or the number of nesting levels of the for-loop.*

Definition 1.2 (Scalar Variable). *A scalar variable is a single element identified by a name, a type and a value.*

Definition 1.3 (Vector Variable). *A vector variable is a multidimensional array of scalar variables of rank R. A single scalar variable can be extracted from a vector variable with the use of a monodimensional array, also called vector of indices vector, of length R.*

Definition 1.4 (Scalar Equation). *A scalar equation is a single mathematical equation that accesses one or more scalar variables.*

Definition 1.5 (Vector Equation). *A vector equation is a nested for-loop of rank R that declares one or more scalar equations. A single equation can be extracted from a vector equation with the use of a monodimensional array, also called inductions vector, of length R.*

Definition 1.6 (Matched Variable)**.** *The matched scalar or vector variable V of, respectively, a scalar or vector equation E is the variable that is designated to be computed from equation E during the simulation.*

Given the Modelica IR generated by the previous pass, the model must be first solved by following a pipeline similar to the one described in section 1.5. Then, its descriptive nature must be converted into an imperative one, so that the system can be simulated in a procedural way. The main difference with respect to other compilers is that the MARCO pipeline is also able to manage vector variables and vector equations. The three main phases for solving a model are matching, strongly connected components resolution and scheduling.

Matching The first step required to solve a model is to find a matched variable for every equation. This means we need to determine which equation determine which variable. When dealing with scalar variables, we need to assign to every variable the scalar equations from which we will compute its value. A bipartite graph matching algorithm can be used for this purpose. Doing so becomes harder when dealing with vector variables and vector equations since we must now match multidimensional variables to multidimensional equations. For this purpose, a new algorithm that is able to preserve for-loops and array structures of the variables as much as possible have been proposed. It was also proven that this problem becomes NP-complete when adding the requirement of maximizing the number of loop cycles preserved in the output of this stage [19]. During this pass, some vector equations may be split into more parts if necessary. This is the case when different indices of a vector equation determine different vector variables. An example of this behavior is shown in listing 1.10. Here, equation *eq2* must be matched partly with x and partly with y, as represented in the bipartite graph of figure 1.7a. To solve this problem, equation *eq2* is split into two equations with the same behavior based on the ranges, each of which is then matched with a different variable, as shown in figure 1.7b. At the end of this pass, every vector equation is associated to the part of the vector variable it determines.

SCC Resolution In this stage, algebraic loops among array variables are found and solved. The search currently consists in applying the Tiernan algorithm [53]. This algorithm, given a directed graph, finds and enumerates all existing SCCs. It is also possible to provide a routine to solve those cycles, which can eventually create more nodes. Since we are using vector equations, when an algebraic loop is found among two equations, the actual cycle may concern only a range of inductions of the vector equations involved. For this reason, before solving every SCC, we “trim” the vector equations so that the scalar

Listing 1.10 Matching example

```
model MatchingExample
  Real[10] x;
  Real[10] y;
equation
  for i in 1:5 loop
    x[i] = 2 * i; // eq1
  end for;

  for j in 1:10 loop
    x[j] = y[j]; // eq2
  end for;

  for k in 6:10 loop
    y[k] = 3 * k; // eq3
  end for;
end MatchingExample;
```

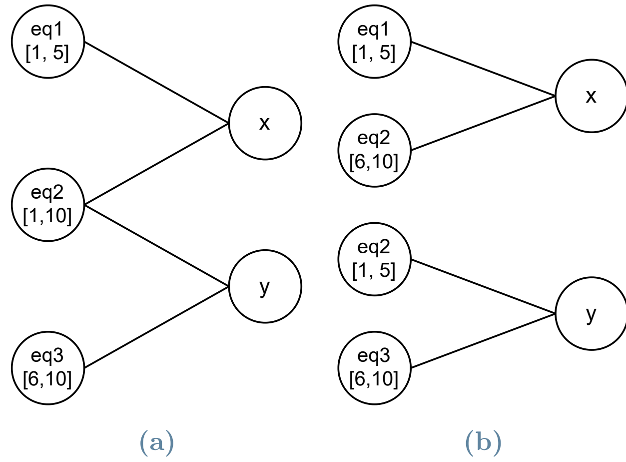


Figure 1.7: Matching example

equations that do not actually take part in the cycle are left untouched by the SCC resolution routine, where the actual cycle is solved by recursively explicitating and replacing equations into each other. An example of this is shown in listing 1.11. Figure 1.8a illustrate the dependency graph of the system, where nodes represent the equations and edges represent the variables from which the nodes depend on. In this example, *eq2* has a cyclic dependency with *eq3*, but only for some of its indices. For this reason, the equation is first split into two as shown in figure 1.8b, so that only the actual cycle is then solved. At the end of this phase the directed graph has become a Directed Acyclic Graph (DAG). This means that we are left with a list of vector equations, each matched with a vector variable, without cyclic dependencies.

Scheduling At this point, since we have removed all SCCs from the DAG dependencies graph, we only need to perform a topological sort of the equations. With this pass, we have obtained the sequential order of vector equations. They can then be transformed into assignment operations, that the simulation is required to perform at every iteration.

State Update At the current state of development of MARCO, the only available solver is the Forward Euler. For this reason, if the model contains any algebraic loops that cannot be solved, we cannot proceed with the simulation. Furthermore, this method has the restrictions of stability and accuracy discussed in section 1.1. In order to update the state of the system, we add to the list of assignment operations also the update of the state variables based on equation 1.4. This must be done for every vector state variable, but each of them can be easily compressed into a for-loop.

Listing 1.11 SCC example

```
model SCCExample
  Real[10] x;
  Real[10] y;
equation
  for i in 1:5 loop
    x[i] = 0; // eq1
  end for;

  for j in 1:10 loop
    y[j] = 4 - x[j]; // eq2
  end for;

  for k in 6:10 loop
    x[k] = y[k] + 2; // eq3
  end for;
end SCCExample;
```

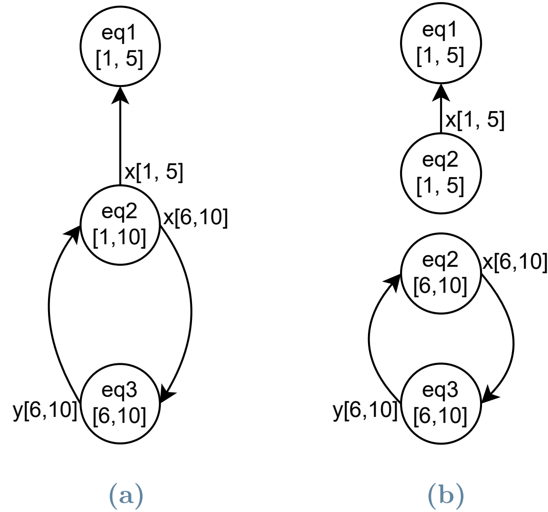


Figure 1.8: SCC example

After the model has been solved, the Modelica IR must be then transformed into LLVM-IR, suitable for generating machine code. As explained in section 1.6, this can be easily done by translating every element of the Modelica dialect into any of the dialects inside the MLIR repository, then calling the transformation passes, already available in the MLIR framework, from these IRs to the LLVM-IR. The whole transformation described in this subsection will be modified in chapter 2 in order to accommodate the DAE solver based on SUNDIALS IDA.

1.7.3. Back-End: Lowering

At this stage, the IR of the compiler is only composed of operations belonging to the LLVM-IR. It can be therefore transformed into assembly code suitable for the desired target architecture. Notice that, differently from the other Modelica compilers described in section 1.5, here we directly generate assembly bitcode, rather than C code which must be then compiled with GCC or CLANG.

2 | Design and Implementation

In this chapter we show the design and implementation procedure utilized for enhancing MARCO with a DAE solver. First an easy to write but rather slow implementation of the DAE solver is shown. Then this DAE solver is optimized by taking advantage of the MARCO pipeline and the MLIR framework. Finally, we describe how to implement the most complex aspects of this procedure.

2.1. Raw DAE Mode

The most simple method to use a DAE solver like IDA is to provide to the solver the entire DAE system in the form of equation 1.1. This way the system $\boldsymbol{F}$ is directly solved with an implicit method like the Newton's iteration and an integration algorithm such as the BDF method. Using a DAE solver, instead of translating the model in ODE form and then using an ODE solver, entails several advantages and disadvantages.

The main advantage of this method is that the computation of derivatives and the integration procedure is done in one single pass. Furthermore, this is done with an implicit algorithm, thus solving implicitly both implicit algebraic equations that might be present and the implicit integration method. This also allows a better step size control when dealing with stiff problems, so that we are not required to use very short time steps. Another advantage is that we can avoid transformations on the model. This means that we could skip entirely some stages of the compiler, such as the matching, the SCC resolution and the scheduling, described in section 1.7.2. This makes it very easy to code, also when pursuing the objective of keeping intact vector equations and variables.

The implementation inside the MARCO compiler will generate code that uses the SUNDIALS IDA library. As explained in section 1.3, in order to do so we must provide the C++ callbacks that compute the residual function and the Jacobian matrix of the entire system $\boldsymbol{F}$. In turn, these are generated from the information contained in the Modelica-IR previously described in section 1.7.1. With this data, we can create two parametric functions, one that computes the residual and one that compute the Jacobian row of a

given vector equation.

The disadvantage of this method, on the other hand, is that it scales poorly. The majority of real-world phenomena that a person might want to simulate usually only contain a small amount, between 5% and 20%, of differential equations and algebraic implicit equations. The remaining ones are only alias equations or linear equations that depend on few variables. These equations do not need any kind of implicit method to be solved, and using such methods would result in a waste of time. In particular, since Newton's method requires the computation of the Jacobian matrix, the algorithm scales quadratically with respect to the number of scalar variables. Even when we leverage sparse algebra, the required time and memory do not justify the usage of an implicit solver for computing linear algebraic equations as well.

2.2. Causalized DAE Mode

As mentioned in the previous section, we want to address the problem of scalability. This can be achieved by combining the interesting aspects of both the DAE-to-ODE translation and the Raw DAE method. The latter is interesting because it solves in one single pass the implicit stiff integration algorithm and the implicit algebraic equations. The former is also interesting because, since the amount of differential and implicit algebraic equations is small in proportion to the whole system, the computed BLT incidence matrix is "almost" an LT matrix. Therefore, we can select and solve with IDA only the BLT blocks that actually need an implicit method.

This can be done by exposing to the DAE solver only the variables that actually need such a solver in order to be computed. We must therefore hide from the solver all variables that we can compute with a series of trivial assignments. The selection of these variable is fairly simple when dealing with scalar variables, but it becomes harder when we add the objective of keeping intact vector equations as much as possible. As a result of this selection, we will divide all variables into two categories: trivial and non-trivial variables.

Definition 2.1 (Trivial Variable). *A trivial variable is a variable matched with an explicitable algebraic equation. This means that the value of such variable can be computed with a single assignment operation after all the other variables on which the matched equation depends have been determined.*

Definition 2.2 (Non-Trivial Variable). *A non-trivial variable is a variable that is not matched with an explicitable algebraic equation. This means that it must be matched with a differential equation, an implicit algebraic equation, or an equation belonging to an*

unsolved algebraic loop. Such variables cannot be computed with an assignment operation and therefore must be computed with a DAE solver.

Definition 2.3 (Trivial Equations)**.** *A trivial equation is an equation matched with a trivial variable. This means that it can be transformed into an assignment operation.*

Definition 2.4 (Non-Trivial Equations)**.** *A non-trivial equation is an equation matched with a non-trivial variable. This means that it cannot be transformed into an assignment operation, but it must be utilized by a DAE solver to compute the matched non-trivial variable.*

Starting from the DAE system $\boldsymbol{F}$ of equation 1.1, we can unpack the vector of algebraic variables as $\boldsymbol{v} = [\boldsymbol{s}\,\boldsymbol{w}]$, where $\boldsymbol{s}$ are the trivial algebraic variables and $\boldsymbol{w}$ are the non-trivial algebraic variables. We can now split the system $\boldsymbol{F}$ into $\hat{\boldsymbol{F}}$ and $\tilde{\boldsymbol{F}}$ as follows:

$$\begin{cases} \hat{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{w}(t), \boldsymbol{u}(t), t) = 0 \\ \boldsymbol{s}(t) = \tilde{\boldsymbol{F}}(\boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \boldsymbol{v}(t), \boldsymbol{u}(t), t) \end{cases} \tag{2.1}$$

where $\hat{\boldsymbol{F}}$ represent the implicit subsystem of the original model that must be solved with the implicit DAE solver, while $\tilde{\boldsymbol{F}}$ is the explicit subsystem that can be trivially solved with a series of assignments. This way all the trivial variables $\boldsymbol{s}$ are hidden from IDA. As explained in the previous section, since only a small amount of variables are non-trivial in a real-world system, the subsystem $\hat{\boldsymbol{F}}$ is usually very small in comparison with $\boldsymbol{F}$ or $\tilde{\boldsymbol{F}}$.

In order to transform the original systems $\boldsymbol{F}$ into the system in the form of equation 2.1, we must perform the following operations. First we must identify the vector $\boldsymbol{y} = [\dot{\boldsymbol{x}}\ \boldsymbol{w}]$ of non-trivial variables that must be passed to IDA. This can be done by modifying the already existing MARCO pipeline and it will be shown how to do so in section 2.3. Then, since some of the equations that must be passed to IDA may depend on variables contained in the vector $\boldsymbol{s}$, we must remove these dependencies. This problem will be solved in section 2.4. Finally, we must generate the callback C++ functions needed by IDA that compute the residual function of the system $\hat{\boldsymbol{F}}$ and its Jacobian matrix $\boldsymbol{J} = \frac{\partial \hat{\boldsymbol{F}}}{\partial \boldsymbol{y}}$. This can be done starting from the vector equations matched with all the variables contained in vector $\boldsymbol{y}$, and it will be shown in section 2.5.

2.3. Identification of Non-Trivial Variables

The most important feature required for an efficient DAE solver is the distinction between trivial and non-trivial variables. To accomplish this, the existing MARCO pipeline has been modified. This new procedure will be explained in both the current section and in the following one with the help of the example – given in implicit DAE form – shown in the system of equations 2.2, together with its corresponding incidence matrix shown in table 2.1. Here, $\boldsymbol{f_i}$ represents the vector equations, $\boldsymbol{x}$ the state variable vector, $\dot{\boldsymbol{x}}$ the derivative variable vector, $\boldsymbol{v_i}$ the algebraic variable vectors, $\boldsymbol{u}$ the input variable vector and t the time variable. Notice that, without losing generality, $\dot{\boldsymbol{x}}$ and all $\boldsymbol{v_i}$ can be multidimensional variables with any rank R, as long as the total number of scalar equations and unknown scalar variables are equal. Doing so, each cell of the IM in table 2.1 is actually a matrix with size $n_i \times n_i$, where n_i is the total dimension of the i-th variable. For the purpose of this example and without losing generality, we assume that all these cells are LT matrices.

$$\begin{cases} \boldsymbol{f_1}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_4}, \boldsymbol{v_5}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_2}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_7}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_3}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_4}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{v_4}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_5}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_1}, \boldsymbol{v_4}, \boldsymbol{v_6}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_6}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_7}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_8}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{f_9}(\boldsymbol{x}, \boldsymbol{v_2}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \end{cases} \tag{2.2}$$

	$\dot{\boldsymbol{x}}$	$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\boldsymbol{v_3}$	$\boldsymbol{v_4}$	$\boldsymbol{v_5}$	$\boldsymbol{v_6}$	$\boldsymbol{v_7}$	$\boldsymbol{v_8}$
$\boldsymbol{f_1}$	0	1	0	0	1	1	0	0	1
$\boldsymbol{f_2}$	0	1	0	0	0	0	0	1	1
$\boldsymbol{f_3}$	0	0	0	1	0	0	0	0	1
$\boldsymbol{f_4}$	0	0	0	1	1	1	0	1	0
$\boldsymbol{f_5}$	1	1	0	0	1	0	1	0	1
$\boldsymbol{f_6}$	0	0	0	1	0	0	0	0	0
$\boldsymbol{f_7}$	1	0	1	1	0	1	0	1	0
$\boldsymbol{f_8}$	0	1	0	1	0	0	0	1	0
$\boldsymbol{f_9}$	0	0	1	0	0	1	0	1	1

Table 2.1: DAE example

2.3.1. Matching

Since we are not introducing any new feature inside the parser, nor any new syntactic sugar, the front-end of MARCO remains unchanged. Likewise, the conversion of the AST to Modelica MLIR remains unmodified, thus leading to the same IR entering the model solving process. Using a DAE solver instead of an Euler solver also does not influence the matching phase. The matching result of the example 2.2 is shown in 2.3. As explained in section 1.7.2, the matching process may split some $\boldsymbol{f_i}$ into multiple vector equations when not all the iteration indices lead to the same matched variable. This would just

create more rows in the incidence matrix, without introducing further complexity within the design. Therefore, for the sake of simplicity, we will assume that no equation gets split during the matching process.

$$\begin{cases} \boldsymbol{f_1}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_4}, \boldsymbol{v_5}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_4}) \\ \boldsymbol{f_2}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_7}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_7}) \\ \boldsymbol{f_3}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_8}) \\ \boldsymbol{f_4}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{v_4}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_5}) \\ \boldsymbol{f_5}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_1}, \boldsymbol{v_4}, \boldsymbol{v_6}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_6}) \\ \boldsymbol{f_6}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_3}) \\ \boldsymbol{f_7}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 & \text{(matched with } \dot{\boldsymbol{x}}) \\ \boldsymbol{f_8}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_1}) \\ \boldsymbol{f_9}(\boldsymbol{x}, \boldsymbol{v_2}, \boldsymbol{v_5}, \boldsymbol{v_7}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 & \text{(matched with } \boldsymbol{v_2}) \end{cases} \tag{2.3}$$

2.3.2. SCC Resolution

In 2.3 we can notice that equation $\boldsymbol{f_2}$ and $\boldsymbol{f_8}$ are respectively matched with variables $\boldsymbol{v_7}$ and $\boldsymbol{v_1}$, but both these variables appear in both equations. The same happens between equations $\boldsymbol{f_1}$ and $\boldsymbol{f_4}$ with variables $\boldsymbol{v_4}$ and $\boldsymbol{v_5}$. This reveals the existence of two algebraic loops within the system. In both cases the matched variable can be chosen randomly between the two options – or with the use of some heuristics. The SCC resolution pass finds all such algebraic loops with the use of the Tiernan algorithm [53] and tries to solve them. For the purpose of this example, we will assume that the cycle between $\boldsymbol{f_2}$ and $\boldsymbol{f_8}$ is solvable while the one between $\boldsymbol{f_1}$ and $\boldsymbol{f_4}$ is not. To solve the first SCC, we only need to make one of the two equation explicit with respect to the matched variable, and then use it to replace all the appearances of the variable within the second equation. This is usually a simple task when dealing with linear equations. The procedure for this specific example is shown in 2.4.

$$\begin{gathered} \boldsymbol{v_7} = \boldsymbol{f_2'}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_8}, \boldsymbol{u}, t) \implies \boldsymbol{f_8}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{v_7}, \boldsymbol{u}, t) \\ = \boldsymbol{f_8}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{f_2'}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_8}, \boldsymbol{u}, t), \boldsymbol{u}, t) = \boldsymbol{f_8'}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{u}, t) = 0 \end{gathered} \tag{2.4}$$

The second SCC is assumed to be unsolvable. This issue usually arises when all the equations composing the SCC are implicit and thus cannot be made explicit. Furthermore, with the use of some heuristics, we may also not want to solve the SCC with the aforementioned procedure if the algebraic loop contains a lot of equations or if these equations

are highly non-linear.

As explained in section 1.1, the Forward Euler method cannot deal with algebraic loop and, if any are encountered, the compilation process must be stopped. When using a DAE solver like IDA, we do not have this restriction any more and we can deal with algebraic loops by exposing these variables to the implicit solver. We can therefore mark $\boldsymbol{v_4}$ and $\boldsymbol{v_5}$ as non-trivial – renaming them to $\boldsymbol{w_4}$ and $\boldsymbol{w_5}$ – so that, later on, they will be used to generate the Residual function and Jacobian matrix of the subsystem that is exposed to IDA. In 2.5 and 2.2 we can see the updated system after this transformation: the SCC has been solved and the variables renamed.

$$
\begin{cases}
\boldsymbol{f_1}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{v_7} = \boldsymbol{f_2}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_8}, \boldsymbol{u}, t) \\
\boldsymbol{f_3}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_4}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_5}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_1}, \boldsymbol{w_4}, \boldsymbol{v_6}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_6}(\boldsymbol{x}, \boldsymbol{v_3}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_7}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{w_5}, \boldsymbol{v_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_8'}(\boldsymbol{x}, \boldsymbol{v_1}, \boldsymbol{v_3}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_9}(\boldsymbol{x}, \boldsymbol{v_2}, \boldsymbol{w_5}, \boldsymbol{v_7}, \boldsymbol{v_8}, \boldsymbol{u}, t) = 0
\end{cases}
\tag{2.5}
$$

	$\dot{\boldsymbol{x}}$	$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\boldsymbol{v_3}$	$\boldsymbol{w_4}$	$\boldsymbol{w_5}$	$\boldsymbol{v_6}$	$\boldsymbol{v_7}$	$\boldsymbol{v_8}$
$\boldsymbol{f_1}$	0	1	0	0	1	1	0	0	1
$\boldsymbol{f_2}$	0	1	0	0	0	0	0	1	1
$\boldsymbol{f_3}$	0	0	0	1	0	0	0	0	1
$\boldsymbol{f_4}$	0	0	0	1	1	1	0	1	0
$\boldsymbol{f_5}$	1	1	0	0	1	0	1	0	1
$\boldsymbol{f_6}$	0	0	0	1	0	0	0	0	0
$\boldsymbol{f_7}$	1	0	1	1	0	1	0	1	0
$\boldsymbol{f_8'}$	0	1	0	1	0	0	0	0	0
$\boldsymbol{f_9}$	0	0	1	0	0	1	0	1	1

Table 2.2: DAE example after SCC resolution

2.3.3. Intermediate Passes

Two simple intermediate transformations have been added between the SCC resolution and the scheduling passes. The first one performs a linear scan of the system, finds all equations matched with a derivative variable and marks them as non-trivial. The second one performs another linear scan of the system and tries to explicitate all the equations with respect to the variable they are matched with. If this is not possible, the variable and the equation are marked as non-trivial. For the purpose of this example we will assume that equation $\boldsymbol{f_3}$ cannot be explicitated with respect to variable $\boldsymbol{v_8}$ and it is therefore marked as non-trivial.

After these two intermediate passes, all the three types of non-trivial variables that must be exposed to IDA have been discovered. We can therefore mark all the remaining variables and equations as trivial. The updated system after this pass, with the equations made explicit and the variables renamed, is shown in 2.6 and 2.3.

$$
\begin{cases}
\boldsymbol{f_1}(\boldsymbol{x}, \boldsymbol{s_1}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_7} = \boldsymbol{f_2'}(\boldsymbol{x}, \boldsymbol{s_1}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\
\boldsymbol{f_3}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_4}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_6} = \boldsymbol{f_5'}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{s_1}, \boldsymbol{w_4}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\
\boldsymbol{s_3} = \boldsymbol{f_6'}(\boldsymbol{x}, \boldsymbol{u}, t) \\
\boldsymbol{f_7}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{s_2}, \boldsymbol{s_3}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_1} = \boldsymbol{f_8''}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{u}, t) \\
\boldsymbol{s_2} = \boldsymbol{f_9'}(\boldsymbol{x}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{w_8}, \boldsymbol{u}, t)
\end{cases}
\tag{2.6}
$$

	$\dot{\boldsymbol{x}}$	$\boldsymbol{s_1}$	$\boldsymbol{s_2}$	$\boldsymbol{s_3}$	$\boldsymbol{w_4}$	$\boldsymbol{w_5}$	$\boldsymbol{s_6}$	$\boldsymbol{s_7}$	$\boldsymbol{w_8}$
$\boldsymbol{f_1}$	0	1	0	0	1	1	0	0	1
$\boldsymbol{f_2'}$	0	1	0	0	0	0	0	1	1
$\boldsymbol{f_3}$	0	0	0	1	0	0	0	0	1
$\boldsymbol{f_4}$	0	0	0	1	1	1	0	1	0
$\boldsymbol{f_5'}$	1	1	0	0	1	0	1	0	1
$\boldsymbol{f_6'}$	0	0	0	1	0	0	0	0	0
$\boldsymbol{f_7}$	1	0	1	1	0	1	0	1	0
$\boldsymbol{f_8''}$	0	1	0	1	0	0	0	0	0
$\boldsymbol{f_9'}$	0	0	1	0	0	1	0	1	1

Table 2.3: DAE example after the intermediate passes

2.3.4. Scheduling

The last pass required to solve the model is the scheduling. After this pass we will have obtained the BLT matrix of the system. The procedure still consists in obtaining the topological ordering of the DAG of the system, where equations represent the nodes, incoming edges represent the variables on which the node depends, and outgoing edges represent the variable determined by the node, – that is, the matched variable. The only different aspect from before is that now we need to deal with the remaining SCCs that arise from unsolved algebraic loops. Since these SCCs will be solved altogether by the DAE solver, we can put together all nodes of an SCC in a single super-node. These nodes, differently from the others, may solve more than one vector variable at once, because the original nodes may be matched with different vector variables.

The DAG of our example is shown in figure 2.1. Following one of the possible topological sorts, we obtain the system shown in equation 2.7. The IM shown in table 2.4 has now become a BLT matrix. Here, the red cells of the BLT matrix represent the subsystems that must be solved by the DAE solver. These subsystems will be called non-trivial BLT blocks.

Definition 2.5 (Non-Trivial BLT Block)**.** *A Non-Trivial BLT Block is a block along the main diagonal of the BLT matrix that cannot be transformed into an assignment operation in order to compute the value of the vector variable it is matched with. It can have size* 1×1 *if the corresponding vector equation is implicit or differential, or it can have size* $m_i \times m_i$ *if it identifies an algebraic loop.*

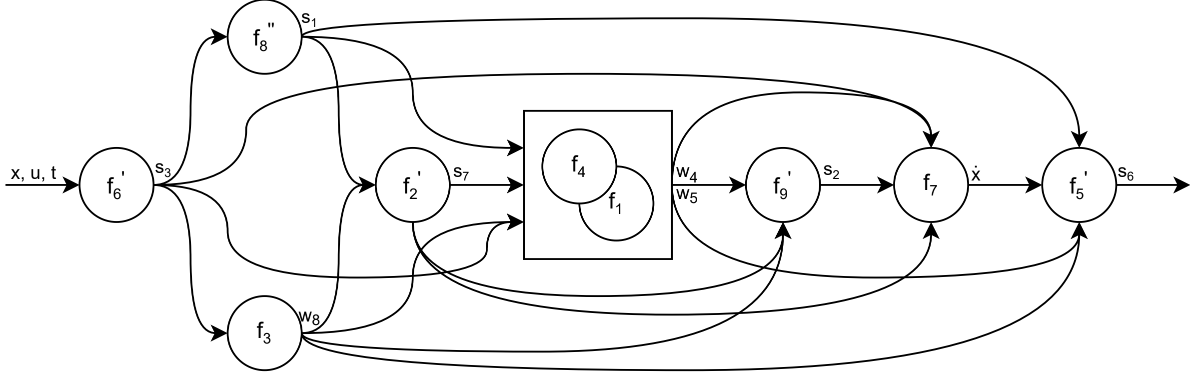


Figure 2.1: DAG of the system example during the scheduling pass

The remaining cells on the BLT main diagonal, colored in green, correspond to the trivial variables that can be computed with a series of assignment operations once all its dependencies are computed. These blocks are always 1×1.

$$
\begin{cases}
\boldsymbol{s_3} = \boldsymbol{f'_6}(\boldsymbol{x}, \boldsymbol{u}, t) \\
\boldsymbol{f_3}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_1} = \boldsymbol{f''_8}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{u}, t) \\
\boldsymbol{s_7} = \boldsymbol{f'_2}(\boldsymbol{x}, \boldsymbol{s_1}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\
\boldsymbol{f_4}(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{f_1}(\boldsymbol{x}, \boldsymbol{s_1}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_2} = \boldsymbol{f'_9}(\boldsymbol{x}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\
\boldsymbol{f_7}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{s_2}, \boldsymbol{s_3}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{u}, t) = 0 \\
\boldsymbol{s_6} = \boldsymbol{f'_5}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{s_1}, \boldsymbol{w_4}, \boldsymbol{w_8}, \boldsymbol{u}, t)
\end{cases}
\tag{2.7}
$$

	$\boldsymbol{s_3}$	$\boldsymbol{w_8}$	$\boldsymbol{s_1}$	$\boldsymbol{s_7}$	$\boldsymbol{w_5}$	$\boldsymbol{w_4}$	$\boldsymbol{s_2}$	$\dot{\boldsymbol{x}}$	$\boldsymbol{s_6}$
$\boldsymbol{f'_6}$	1	0	0	0	0	0	0	0	0
$\boldsymbol{f_3}$	1	1	0	0	0	0	0	0	0
$\boldsymbol{f''_8}$	1	0	1	0	0	0	0	0	0
$\boldsymbol{f'_2}$	0	1	1	1	0	0	0	0	0
$\boldsymbol{f_4}$	1	0	0	1	1	1	0	0	0
$\boldsymbol{f_1}$	0	1	1	0	1	1	0	0	0
$\boldsymbol{f'_9}$	0	1	0	1	1	0	1	0	0
$\boldsymbol{f_7}$	1	0	0	1	1	0	1	1	0
$\boldsymbol{f'_5}$	0	1	1	0	0	1	0	1	1

Table 2.4: DAE example after scheduling

2.4. Trivial Variable Substitution

Once obtained the final BLT IM of the original system, we must extrapolate from it the subsystem that must be computed by the DAE solver and the subsystem that must be computed through a series of assignments. This can be done by removing from all non-trivial equations the dependencies from trivial variables. Since all trivial equations are already explicitated, it is only necessary to recursively substitute all trivial variable accesses inside non-trivial equations with the right-hand side of the equation that is matched with such variables.

Definition 2.6 (Variable Access)**.** *A Variable Access is a reference to a scalar variable within a vector variable of rank R given a monodimensional array, also called indices vector, of length R.*

Particular care must be performed if more than one equation is matched with the same trivial variable. In this case the equation inside a non-trivial BLT block that uses this variable may be split into more instances, each of which iterates across a different set of indices, and the variable is then replaced with different trivial equations. For example, in the model of a one-dimensional rod described in listing 1.4, the equation containing the derivative of T, thus a non-trivial equation, makes use of the trivial variables $Qleft$ and $Qright$. These two trivial variables are matched to different equations depending on the index. For this reason, the non-trivial equation, with original length n, will be split during this pass into three equations, with total length n. The Jacobian matrix required by IDA will be later computed using only these three equations, independently from the value of n:

$$\begin{cases} rho * V * C * der(T[1]) = [-h * A_s * (T[1] - Tamb)] \\ \qquad + [-k * A_c * (T[1] - T[2])/(L/n)] \\ rho * V * C * der(T[i]) = [-h * A_s * (T[i] - Tamb)] \\ \qquad + [-k * A_c * (T[i] - T[i-1])/(L/n)] \\ \qquad + [-k * A_c * (T[i-1] - T[i])/(L/n)] \quad \forall i \in 2, \ldots, n-1 \\ rho * V * C * der(T[n]) = [-h * A_s * (T[n] - Tamb)] \\ \qquad + [-k * A_c * (T[n] - T[n-1])/(L/n)] \end{cases} \tag{2.8}$$

Algorithm 2.1 shows how to perform this operation. Given the list of trivial equations $trivialEquations$ and the list of non-trivial equations $nonTrivialEquations$, every trivial variable access within every non-trivial equation is found. For each one of these accesses, the non-trivial equation is cloned as many times as the number of equations matched with that variable access. Then that variable access is replaced with the right-hand side of the equation matched with the trivial variable. This process is repeated until the non-trivial equation only contains non-trivial variable accesses and constants. The algorithm also makes use of some utility methods, which depend on the IR. Some of them are self-explanatory, while the behavior of the remaining ones is reported here for clarity:

- **areDisjoint(**acc_i**,** acc_j**):** Returns true if the two given accesses have no range of indices in common, false otherwise.

- **clone(eq):** Returns a clone of the equation eq.
- **getAccessedVar(acc):** Returns the variable referenced by the variable access acc.
- **getAllVarAccesses(eq):** Returns a list of all variable accesses inside the equation eq.
- **getMatchedVar(eq):** Returns the matched variable of the equation eq.
- **getMatchedAccess(eq):** Returns the variable access of the matched variable of the equation eq.
- **intersection(acc_i, acc_j):** Returns the range of indices in common between the two given accesses.
- **replaceUses(eq_s, eq_d):** Returns the destination equation eq_d where all accesses to the variable matched with the source equation eq_s are replaced with the right-hand side of eq_s.
- **setInductions(eq, $inds$):** Returns the equation eq with its induction ranges changed to $inds$.

Algorithm 2.1: Trivial variable substitution

```
1  Function trivialVariableSubstitution
2      Input: trivialEquations, nonTrivialEquations
3      trivialEqMap ← ∅
4      foreach eq_i ∈ trivialEquations do
5          mVar ← getMatchedVar(eq_i)
6          trivialEqMap[mVar] ← append(trivialEqMap[mVar], eq_i)
7      end
8      foreach eq_i ∈ nonTrivialEquations do
9          foreach varAcc_j ∈ getAllVarAccesses(eq_i) do
10             var ← getAccessedVar(varAcc_j)
11             if ¬empty(trivialEqMap[var]) then
12                 foreach eq_k ∈ trivialEqMap[var] do
13                     mVarAcc ← getMatchedAccess(eq_k)
14                     if ¬areDisjoint(varAcc_j, mVarAcc) then
15                         newEq ← clone(eq_i)
16                         newInductions ← intersection(varAcc_j, mVarAcc)
17                         newEq ← setInductions(newEq, newInductions)
18                         newEq ← replaceUses(eq_k, newEq)
19                         nonTrivialEquations ← append(nonTrivialEquations, newEq)
20                     end
21                 end
22                 nonTrivialEquations ← erase(nonTrivialEquations, eq_i)
23             end
24         end
25     end
26 end
```

After applying this algorithm, all non-trivial equations are now independent from trivial variables. For this reason, we can reorder the system, and consequently also the IM, so

that all non-trivial equations come before the trivial equations. This must of course be done by maintaining the relative order inside the two set of equations. Doing so we obtain the system shown in equation 2.9 and the IM in table 2.5.

$$\begin{cases} f_3'(\boldsymbol{x}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\ f_4'(\boldsymbol{x}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{u}, t) = 0 \\ f_1'(\boldsymbol{x}, \boldsymbol{w_4}, \boldsymbol{w_5}, \boldsymbol{w_8}, \boldsymbol{u}, t) = 0 \\ f_7'(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{w_5}, \boldsymbol{u}, t) = 0 \\ \boldsymbol{s_3} = f_6'(\boldsymbol{x}, \boldsymbol{u}, t) \\ \boldsymbol{s_1} = f_8''(\boldsymbol{x}, \boldsymbol{s_3}, \boldsymbol{u}, t) \\ \boldsymbol{s_7} = f_2'(\boldsymbol{x}, \boldsymbol{s_1}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\ \boldsymbol{s_2} = f_9'(\boldsymbol{x}, \boldsymbol{w_5}, \boldsymbol{s_7}, \boldsymbol{w_8}, \boldsymbol{u}, t) \\ \boldsymbol{s_6} = f_5'(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{s_1}, \boldsymbol{w_4}, \boldsymbol{w_8}, \boldsymbol{u}, t) \end{cases} \tag{2.9}$$

	w_8	w_5	w_4	$\dot{x}$	s_3	s_1	s_7	s_2	s_6
f_3'	1	0	0	0	0	0	0	0	0
f_4'	0	1	1	0	0	0	0	0	0
f_1'	0	1	1	0	0	0	0	0	0
f_7'	0	1	0	1	0	0	0	0	0
f_6'	0	0	0	0	1	0	0	0	0
f_8''	0	0	0	0	1	1	0	0	0
f_2'	1	0	0	0	0	1	1	0	0
f_9'	1	1	0	0	0	0	1	1	0
f_5'	1	0	1	1	0	0	0	0	1

Table 2.5: DAE example after trivial variable substitution

The IM obtained in this way is always characterized by four submatrices. The submatrix on the top left is the BLT matrix containing only differential equations, implicit equations and algebraic loops. It is therefore the subsystem $\hat{\boldsymbol{F}}$ of equation 2.1 that must be computed with the DAE solver. The submatrix on the bottom left is not a triangular matrix and contains the dependencies of the trivial equation from the non-trivial variables. The submatrix on the bottom right is always an LT matrix that contains all the remaining trivial equations of the system. It represent the trivial subsystem $\tilde{\boldsymbol{F}}$. Once the non-trivial variables have been solved by IDA, this whole subsystem can be computed with a series of assignment operations. Finally, the submatrix on the top right always contains only zeros.

Now that the subsystem that requires a DAE solver has been extrapolated from the original model, we can use its equations to create the callback functions that compute its Residual function and Jacobian matrix.

2.5. Sundials IDA Integration

SUNDIALS IDA is the DAE solver chosen for the purposes of this project. Its main characteristics were already described in section 1.3 and we present here some further design and implementation details on how the usage of this library was integrated inside MARCO. IDA was added to the pre-existing runtime library, with suitable wrapper func-

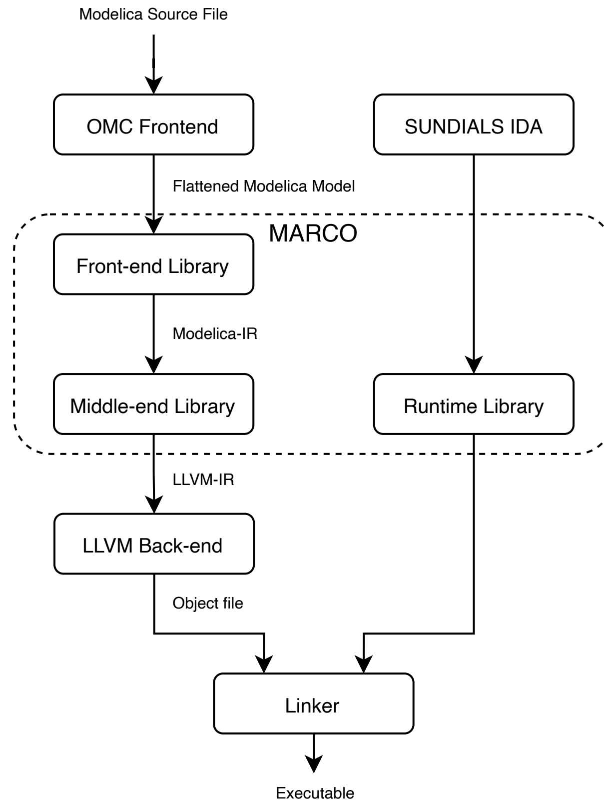


Figure 2.2: MARCO pipeline with SUNDIALS IDA

tions that serve as an interface adapter with the generated simulation code. We chose to not generate code that uses the functions exported by IDA directly for two main reasons. First, this code generation task required the re-definition of any data structure used by IDA into MLIR. This process is difficult, error prone and the implementation needs to be changed after every update of the IDA library. The second reason is that in this way the front, middle and back ends of MARCO only depend on LLVM and MLIR, while the runtime library does not depend on LLVM but only on SUNDIALS IDA. Using an adapter pattern to decouple the IDA interface from the MARCO compiler makes it easy and maintainable to add a further solver, or change an existing one, without having to propagate the changes inside the compiler itself. The new layout of MARCO is shown in figure 2.2.

Inside the middle-end of MARCO, after the model has been solved as described in sections 2.3 and 2.4, the last pass that needs to be performed is the introduction of calls to the IDA library inside the simulation program. A first set of calls must be performed during the initialization of the simulation which allocate and provide to IDA the required information about the model. Then, during the main loop of the simulation, a single

call to IDA must be performed at each iteration so that the DAE solver can compute the state of the system at the next requested time point. Finally, at the end of the simulation, all the used data is deallocated after, eventually, printing some statistics related to the simulation, such as the number of steps taken and the number of Jacobian evaluations. All means of interactions between the main program and IDA inside the runtime library are performed through the use of opaque pointers in order to maintain the state of the solver without exposing implementation details to the main program.

2.5.1. MLIR IDA Dialect

A second MLIR dialect, called IDA dialect, has been introduced inside MARCO to interact with IDA in the runtime library. This dialect depends on the already implemented Modelica dialect and does not add any particularly new syntax or semantic, but it allows to write a more readable and maintainable IR during the various optimization passes of the middle-end.

Most of the new operations introduced consist of simple wrappers around calls to the IDA runtime library. Among the other operations, two in particular are used to build callback functions that compute the residuals and the Jacobian rows of the system starting from a vector equation of the Modelica dialect. These two operations will be discussed in greater details in the next two subsections. Finally, some utility operations were introduced to manage implementation details such as pointer management and addresses of functions. All these operations are validated in-between passes through type checking of their arguments in order to prevent incorrect uses.

2.5.2. Residual Function Computation

Starting from a vector equation of the Modelica dialect, we want to create a parametric function that computes its residual error. This function will have the signature described in listing 2.1, so that it can be computed starting from the time variable, the variable and derivative vectors, and the indices that identify a single scalar equation inside the vector equation. With this approach we will have one residual function for every non-trivial vector equation. Being it also parametric, this approach also leaves room to parallelization of the computation of the entire residual vector of the system, because all computations are independent from each other.

The algorithm that creates the residual function starting from a vector equation is described in algorithm 2.2. *ComputeExpression* is a recursive method that perform a depth first search on the syntax tree of a given equation. It replaces the variables and indices

Listing 2.1 C++ prototype of the parametric Residual function of a vector equation

```
realtype residualFunction_i(
        realtype tt, // Current time value.
        realtype* yy, // Variables vector.
        realtype* yp, // Derivatives vector.
        indextype* ind // Indices vector of the vector equation.
);
```

used inside the equation with the correct arguments of the function. The algorithm also makes use of some utility methods, the most important of which are reported here for clarity:

- **computeVariableOffset**(acc, ind)**:** Computes the flattened scalar index value of the variable accessed by acc starting from the indices of the equation ind and how the variable is accessed acc.
- **createResidualFunction**(str)**:** Create a new function inside the program with name str and with the arguments of the prototype in listing 2.1.
- **getArgs**$(func)$**:** Returns the arguments of the function $func$.
- **insertOp**$(func$, op, $args\ldots)$**:** Insert inside the function $func$ an operation with the same type of op and arguments $args$.
- **insert**$*$**Op**$(func$, $args\ldots)$**:** Insert inside the function $func$ an operation of type $*$ with arguments $args$.
- **isDerivativeVariable**(var)**:** Checks if the variable var is a derivative of order one or not.

2.5.3. Jacobian Matrix Computation

Similarly to residual functions, a parametric function that computes the Jacobian matrix of the system starting from a vector equation is created. This function will have the signature described in listing 2.2, so that a single cell of the Jacobian matrix can be computed starting from the time variable, the variable and derivative vectors, the indices that identify a row of the Jacobian, the variable with respect to which we are making the derivative operation, thus identifying a single cell within the row, and the parameter alpha of equation 1.13. With this approach we will have one Jacobian function for every non-trivial vector equation. Being it also parametric, this approach also leaves room to parallelization of the computation of the entire Jacobian matrix of the system, because every cell can be computed independently from each other.

The algorithm that creates the Jacobian function starting from a vector equation is de-

Algorithm 2.2: Residual function computation

```
1  Function residualFunctionComputation
2      Input: functionName, equation
3      Output: residualFunction
4      residualFunction ← createResidualFunction(functionName)
5      lhs ← computeExpression(residualFunction, getLHS(equation))
6      rhs ← computeExpression(residualFunction, getRHS(equation))
7      resultOp ← insertSubOp(residualFunction, rhs, lhs)
8      insertReturnOp(residualFunction, resultOp)
9      return residualFunction
10 end

11 Function computeExpression
12     Input: function, expression
13     Output: resultValue
14     {time, varVector, derVector, indices} ← getArgs(function)
15     if isConstant(expression) then
16         return insertConstantOp(function, getConstant(expression))
17     end
18     if isInduction(expression) then
19         return indices[getInduction(expression)]
20     end
21     if isTime(expression) then
22         return time
23     end
24     if isVariableAccess(expression) then
25         variableAccess ← getVariableAccess(expression)
26         offset ← computeVariableOffset(variableAccess, indices)
27         if isDerivativeVariable(getAccessedVar(variableAccess)) then
28             return insertAccessOp(function, derVector, offset)
29         end
30         return insertAccessOp(function, varVector, offset)
31     end
32     assert isOperation(expression)
33     childOps ← {}
34     foreach child_i ∈ getOperation(expression) do
35         childOps ← append(childOps, computeExpression(function, child_i))
36     end
37     return insertOp(function, getOperation(expression), childOps)
38 end
```

Listing 2.2 C prototype of the parametric Jacobian function of a vector equation

```
int jacobianFunction_i(
        realtype tt, // Current time value.
        realtype* yy, // Variables vector.
        realtype* yp, // Derivatives vector.
        indextype* ind, // Indices vector of the vector equation.
        indextype var, // Index of the variable with respect to ↩
           which the vector equation is differentiated.
        realtype cj // Alpha parameter.
);
```

scribed in algorithm 2.3. *ComputeDerExpression* is a recursive method that perform a depth first search on the syntax tree of a given equation and computes its symbolic derivative. This method utilizes the already available automatic differentiation of MARCO [49]. It replaces constants, indices and time variables with a zero. Variables are replaced with a one, and derivatives with the alpha parameter if they correspond to the variable in the argument, zero otherwise. Doing so we only compute the derivative once, instead of twice: with respect to a variable, and with respect to its derivative, as shown in 1.13. This can be done by taking advantage of the chain rule of derivation. It is in fact possible to "push" the derivative of an expression along its syntax tree using the chain rule, and only at the end derivate the leaf nodes with respect to both a given variable and its derivative. In a more formal way:

Theorem 2.1. *Starting from equation 1.13:*

$$J = \frac{\partial F}{\partial y} + \alpha \frac{\partial F}{\partial \dot{y}} = \left(\frac{\partial}{\partial y} + \alpha \frac{\partial}{\partial \dot{y}} \right) (F) \tag{2.10}$$

Defining the new operator $\boldsymbol{\partial}^{\boldsymbol{\alpha}}$ *as:*

$$\boldsymbol{\partial}^{\boldsymbol{\alpha}} := \frac{\partial}{\partial y} + \alpha \frac{\partial}{\partial \dot{y}} \tag{2.11}$$

We can obtain, through the chain rule, that:

$$\boldsymbol{\partial}^{\boldsymbol{\alpha}} \left(F \left(G \left(y, \dot{y} \right) \right) \right) = \dot{F} \left(G \left(y, \dot{y} \right) \right) \cdot \boldsymbol{\partial}^{\boldsymbol{\alpha}} \left(G \left(y, \dot{y} \right) \right) \tag{2.12}$$

Proof.

$$
\begin{aligned}
\boldsymbol{\partial}^{\boldsymbol{\alpha}}\left(F\left(G\left(y,\dot{y}\right)\right)\right) &= \left(\frac{\partial}{\partial y}+\alpha\frac{\partial}{\partial \dot{y}}\right)\left(F\left(G\left(y,\dot{y}\right)\right)\right) \\
&= \frac{\partial F\left(G\left(y,\dot{y}\right)\right)}{\partial y}+\alpha\frac{\partial F\left(G\left(y,\dot{y}\right)\right)}{\partial \dot{y}} \\
&= \dot{F}\left(G\left(y,\dot{y}\right)\right)\cdot\frac{\partial G\left(y,\dot{y}\right)}{\partial y}+\alpha\dot{F}\left(G\left(y,\dot{y}\right)\right)\cdot\frac{\partial G\left(y,\dot{y}\right)}{\partial \dot{y}} \\
&= \dot{F}\left(G\left(y,\dot{y}\right)\right)\cdot\left(\frac{\partial G\left(y,\dot{y}\right)}{\partial y}+\alpha\frac{\partial G\left(y,\dot{y}\right)}{\partial \dot{y}}\right) \\
&= \dot{F}\left(G\left(y,\dot{y}\right)\right)\cdot\left(\frac{\partial}{\partial y}+\alpha\frac{\partial}{\partial \dot{y}}\right)\left(G\left(y,\dot{y}\right)\right) \\
&= \dot{F}\left(G\left(y,\dot{y}\right)\right)\cdot\boldsymbol{\partial}^{\boldsymbol{\alpha}}\left(G\left(y,\dot{y}\right)\right)
\end{aligned}
\tag{2.13}
$$

□

The symbolic derivation may generate many sub-expressions that always amount to zero. For this reason, a simple constant folding pass was applied to the resulting functions to eliminate useless computations. The shown algorithm makes use of some utility methods, the ones not already mentioned are reported here:

- **createJacobianFunction(*str*):** Create a new function inside the program with name *str* and with the arguments of the prototype in listing 2.2.
- **derivate(*op*, *children*, *ders*):** Symbolically computes the derivative of operation *op* and maps its children with the correct computed sub-expression of the children *children* and their derivatives *ders*.

2.5.4. Initialization of IDA Data

Once everything required by IDA concerning the model has been computed at compile time, all this data must be passed at runtime to the external library at the start of the simulation. During this initialization phase of IDA, several optional parameters can also be set to change the behavior of the solver, all of which can be chosen by a user of MARCO. Such parameters consist in:

- **Start time** and **End time:** Two real numbers that symbolize the initial and final time instants of the simulation. The end time must be strictly greater than the start time.
- **Equidistant time grid** and **Time step:** The first is a boolean that allows to print the output variables at equally distant time intervals, computed through interpolation

Algorithm 2.3: Jacobian matrix computation

```
1  Function jacobianMatrixComputation
2      Input: functionName, equation
3      Output: jacobianFunction
4      jacobianFunction ← createJacobianFunction(functionName)
5      lhs ← computeDerExpression(jacobianFunction, getLHS(equation))
6      rhs ← computeDerExpression(jacobianFunction, getRHS(equation))
7      resultOp ← addSubOp(jacobianFunction, rhs, lhs)
8      addReturnOp(jacobianFunction, resultOp)
9      return jacobianFunction
10 end

11 Function computeDerExpression
12     Input: function, expression
13     Output: resultValue
14     {time, varVector, derVector, indices, alpha, derVar} ← getArgs(function)
15     zeroOp ← insertConstantOp(function, 0.0)
16     if isConstant(expression) or isInduction(expression) or isTime(expression) then
17         return zeroOp
18     end
19     if isVariableAccess(expression) then
20         variableAccess ← getVariableAccess(expression)
21         offset ← computeVariableOffset(variableAccess, indices)
22         conditionOp ← insertEqOp(function, offset, derVar)
23         if isDerivativeVariable(getAccessedVar(variableAccess)) then
24             return insertIfThenElseOp(function, conditionOp, alpha, zeroOp)
25         end
26         oneOp ← insertConstantOp(function, 1.0)
27         return insertIfThenElseOp(function, conditionOp, oneOp, zeroOp)
28     end
29     assert isOperation(expression)
30     childOps ← {}
31     derChildOps ← {}
32     foreach child_i ∈ getOperation(expression) do
33         childOps ← append(childOps, computeExpression(function, child_i))
34         derChildOps ← append(derChildOps, computeDerExpression(function, child_i))
35     end
36     return derivate(getOperation(expression), childOps, derChildOps)
37 end
```

(option set), or at every step taken by the DAE solver (option not set). The second is a real number that sets the time interval of the output, if the equidistant time grid mode is chosen.

- **Relative tolerance** and **Absolute tolerance:** Two real numbers that control the relative and absolute errors committed at each individual time step. They should not be greater than 10^{-3}, otherwise the results will be highly inaccurate, nor smaller than 10^{-15}, since the error would become comparable to the unit roundoff of the machine arithmetic. The second parameter is only used if the value of a variable is very small, since in this case the relative error grows to infinity. As it will be shown in section 3.2, these two parameters are particularly important in determining the precision of the simulation output.

Also, several other pieces of information concerning the given model, needed for its simulation, are computed and passed to the runtime IDA library. All this data is computed from the model obtained after the pass described in section 2.4 and it includes the following parameters:

- **Number of scalar equations (NEQ):** Integer number symbolizing the total size of the non-trivial variable vectors. It is needed to correctly size the internal vectors needed by IDA.
- **Number of non-zero values (NNZ):** Integer number symbolizing the number of useful cells of the Jacobian matrix of the system. Since systems in the real-world are usually highly sparse, this parameter is required to take advantage of sparse algebra computations. Furthermore, the Jacobian matrix is computed and stored in compressed-sparse-row (CSR) format to further reduce the needed space.
- **Residual callback functions:** List of functions with the signature of listing 2.1, used to compute the residual error of the system through the IDA function in listing 1.1, computed as explained in section 2.5.2.
- **Jacobian callback functions:** List of functions with the signature of listing 2.2, used to compute the Jacobian matrix of the system through the IDA function in listing 1.2, computed as explained in section 2.5.3.
- **Dimensions of the variables:** List of the sizes of every dimension of each vector variable.
- **Dimensions of the equations:** List of the sizes of every range of inductions of each vector equation.
- **Variable accesses inside equations:** List of variables that are accessed inside every vector equation.

The last three lists are needed to compute the indices required by the parametric resid-

ual and Jacobian callback functions, and to compute the coordinates of the cells of the Jacobian matrix that may contain non-zero values. Notice that the size of all this data depends only on the number of vector variables and vector equations and not on the scalar size of such vectors.

2.6. MARCO Main Program

The resulting executable can finally be built. The algorithm 2.4 shows the main of the final produced executable. After the allocation and initialization of the required variables and IDA data, a step of the DAE solver is performed, along with the printing of the desired variables, until the requested end time is reached. Finally, all variables and IDA data are gracefully deallocated.

Algorithm 2.4: Main of the produced MARCO executable using IDA

```
1  Function main
2      Input: startTime, endTime
3      variables ← initVariables()
4      idaData ← initIdaData()
5      currentTime ← startTime
6      printVariables(currentTime, variables)
7      while currentTime < endTime do
8          currentTime ← idaStep(idaData)
9          updateTrivialVariables(variables)
10         printVariables(currentTime, variables)
11     end
12     deinitIdaData(idaData)
13     deinitVariables(variables)
14 end
```

3 | Experimental Results

In this chapter we will demonstrate the usefulness of the adopted approach. First, we will show the correctness of the methodology by comparing the simulation results produced by the executable generated by MARCO with the results produced by simulations from other production-grade compilers. Then we will show that the we have reached the objective of achieving constant compilation time and constant executable binary size with respect to the dimensions of internal vectors. Finally, we will evaluate the execution time and the memory usage of the generated binaries.

3.1. Thermal Chip DAE Model

Several benchmark suites [9, 20] already exist for benchmarking large-scale Modelica models, but they either do not allow scaling through parameters or they make use of advanced features of the Modelica language – such as records – that are not yet supported by MARCO. For this reason, a newly born benchmark suite called HiPerMod [2, 38], developed by our research group, has been used to evaluate the performance of our new approach. The goal of HiPerMod is to provide the means to quantify how far the currently available Modelica compilers are from producing executables with performance comparable to hand-written simulation code. All the models within this suite are made in such a way that their size (in terms of number of equations) can be easily scaled by modifying some parameters without becoming unrealistic.

Among the available benchmarks, the ThermalChipDAE model performs a thermal simulation similar to a finite element analysis. This is the model used for our measurements. The Modelica source file describes a cube-shaped chip made of silicon, where a constant power is continuously applied, starting from an initial time t_0, on half of the bottom surface of the chip. The temperature across the volume of the cube has been discretized into a three-dimensional matrix with respect to three parameters M, N and P. These three parameters, which influence the number of variables within the system, are the ones that will be modified in order to evaluate the scaling capabilities of both our compiler and the generated simulation. Notice that the modification of these parameters does not affect

M	N	P	States	Equations	
4	4	4	64	320	
5	5	5	125	600	
6	6	6	216	1 008	
8	8	8	512	2 304	
10	10	10	1 000	4 400	
13	13	13	2 197	9 464	
16	16	16	4 096	17 408	
20	20	20	8 000	33 600	
25	25	25	15 625	65 000	
32	32	32	32 768	135 168	
40	40	40	64 000	262 400	← OMC simulation up to here
51	51	51	132 651	541 008	← OMC compilation up to here
64	64	64	262 144	1 064 960	
81	81	81	531 441	2 152 008	← MARCO/C++ simulation up to here
102	102	102	1 061 208	4 286 448	
128	128	128	2 097 152	8 454 144	

Table 3.1: Parameters used for the OMC, MARCO and C++ simulations

the physical dimension of the chip, but they only change the number of sub-cubes the original chip is being divided into. This allows to always simulate a feasible real-world system. The full code is available in appendix A.

The model was simulated with both MARCO and OMC using different values for M, N and P. Then, the results were compared with a third batch of simulations of the same model, written in C++, to verify the efficiency of our solution with respect to hand-written and optimized simulation code. The tests started from a small model with only 64 scalar state variables and 320 scalar equations. Then, the number of state variables was doubled at each iteration until a model with 531 441 state variables and 2 152 008 equations was reached. For MARCO and C++, this limit was only imposed by the memory usage of the simulations, which exceeds what is available on our test machine. On the other hand, there is virtually no compile-time resource limit that bounds the size of the model. OMC, on the other hand, was only able to compile models with up to 541 008 equations and simulate models only up to 262 400 equations. This is because, as it will be shown in section 3.6, either the simulation produced by OMC runs out of memory, or the OMC compiler itself takes too long to execute or produces executables too large. The parameters used in the various simulation are shown in table 3.1. All simulations were performed from $t_0 = 0.0$ seconds to $t_n = 1.0$ seconds.

When using OMC, the model was compiled and simulated after the three parameters were changed in the source code. When using MARCO, a first step of flattening of the model, done through OMC, was also required to eliminate some object-oriented features of the

model. Then the resulting flattened model was compiled and simulated. On the contrary, the compilation of the C++ benchmark was done once for all sizes of the model, since the parameters M, N and P are set at run-time through command-line parameters. All the three simulations described were benchmarked both in terms of execution time and memory usage.

The experiments were performed on a Linux machine with the following hardware and software configuration:

- Operating System: Ubuntu 20.04.3 LTS
- CPU: Intel Xeon CPU E5-2650 v3, 2.30GHz, 20 cores
- RAM: 72 GB
- LLVM: 13.0.0
- OpenModelica: v1.19.0 dev-407-g0aeaad6

3.2. Result Correctness

We first want to show that the executable produced by MARCO performs a correct simulation of the model while exploiting the DAE solver. As mentioned in section 2.5.4, the tolerance parameters used by IDA are strictly related to the precision of the result since they determine the acceptable error of the simulation. In particular, using greater tolerance parameters will produce results faster but with a lower accuracy, while using smaller tolerances will produce a more accurate result at the expenses of simulation time.

For this reason, all simulations – OMC, MARCO and C++ – were performed using a default value of 10^{-6} for both the relative and absolute tolerances. Then, the three simulation outputs were compared with the output of a simulation compiled with OMC using tolerances set to 10^{-10}. This way we can evaluate the errors made by MARCO and C++ through a comparison with the error that OMC makes among its two simulations when using different tolerance values. All temperatures of every sub-cube are compared at equidistant time intervals of 0.02 seconds with the more accurate OMC simulation. For each simulation, only the maximum error is reported. These results can be found in table 3.2.

As can be seen, the maximum relative error is slightly above the tolerance parameter, while the maximum absolute error is of two order of magnitude greater. For the relative error, this is because it accumulates through the simulation at every step, thus eventually becoming greater than the tolerance parameter. For the absolute one, it is way greater than the given tolerance because only the relative error is used – as the temperature of

	Maximum Relative Error			Maximum Absolute Error		
States	**OMC**	**MARCO**	**C++**	**OMC**	**MARCO**	**C++**
64	1.61754e-06	1.61755e-06	1.61755e-06	5.09364e-04	5.09364e-04	5.09364e-04
125	3.01230e-06	3.01232e-06	3.01232e-06	9.43988e-04	9.43992e-04	9.43992e-04
216	2.39593e-06	2.39595e-06	2.39595e-06	7.51148e-04	7.51154e-04	7.51154e-04
512	2.29865e-06	2.29864e-06	2.29864e-06	7.21292e-04	7.21292e-04	7.21292e-04
1000	2.32801e-06	2.32800e-06	2.32800e-06	7.38989e-04	7.38987e-04	7.38987e-04
2197	7.18301e-07	7.18302e-07	7.18302e-07	2.48307e-04	2.48307e-04	2.48307e-04
4096	1.98458e-06	1.98457e-06	1.98457e-06	6.27600e-04	6.27597e-04	6.27597e-04
8000	1.65806e-06	1.65809e-06	1.65809e-06	5.24436e-04	5.24446e-04	5.24446e-04
15625	9.15956e-07	9.16007e-07	9.16007e-07	2.89711e-04	2.89727e-04	2.89727e-04
32768	1.56670e-06	1.56688e-06	1.56688e-06	4.96443e-04	4.96500e-04	4.96500e-04
64000	8.54597e-07	8.54597e-07	8.54597e-07	2.96411e-04	2.96411e-04	2.96411e-04

Table 3.2: Absolute errors made by OMC, MARCO and C++ using tolerances equal to 10^{-6} against OMC using tolerances equal to 10^{-10}

the cube does not tend to zero Kelvin. In any case, the errors made by MARCO and C++ are the same as those made by OMC, for this reason we can reasonably conclude that the resulting errors do not significantly depend on the compiler but rather on the DAE solver. Furthermore, both these errors progressively decrease as expected when using smaller tolerance values.

Figure 3.1 shows the resulting temperatures of the thermal chip across a simulation using any of the three compilers with $M = 10$, $N = 10$ and $P = 10$, sampled at the center of the cube and on some of the corners and surfaces.

3.3. Binary Size

In this section we demonstrate that the objective of creating an executable which size does not depend on the length of internal vectors has been achieved. As can be seen in figure 3.2, the size of the bitcode produced by MARCO is constant at around 100 KB, with only minor fluctuations, probably caused by LLVM optimizations. Instead, the OMC binary size grows linearly with the number of variables used, reaching almost 1 GB of size at 132 651 states. This is due to the complete de-vectorization of variables and the loop unrolling that OMC performs at compile time as explained in section 1.5. Finally, C++ also has a constant binary size, which is slightly lower than MARCO and without any fluctuation because it is compiled only once for all simulations.

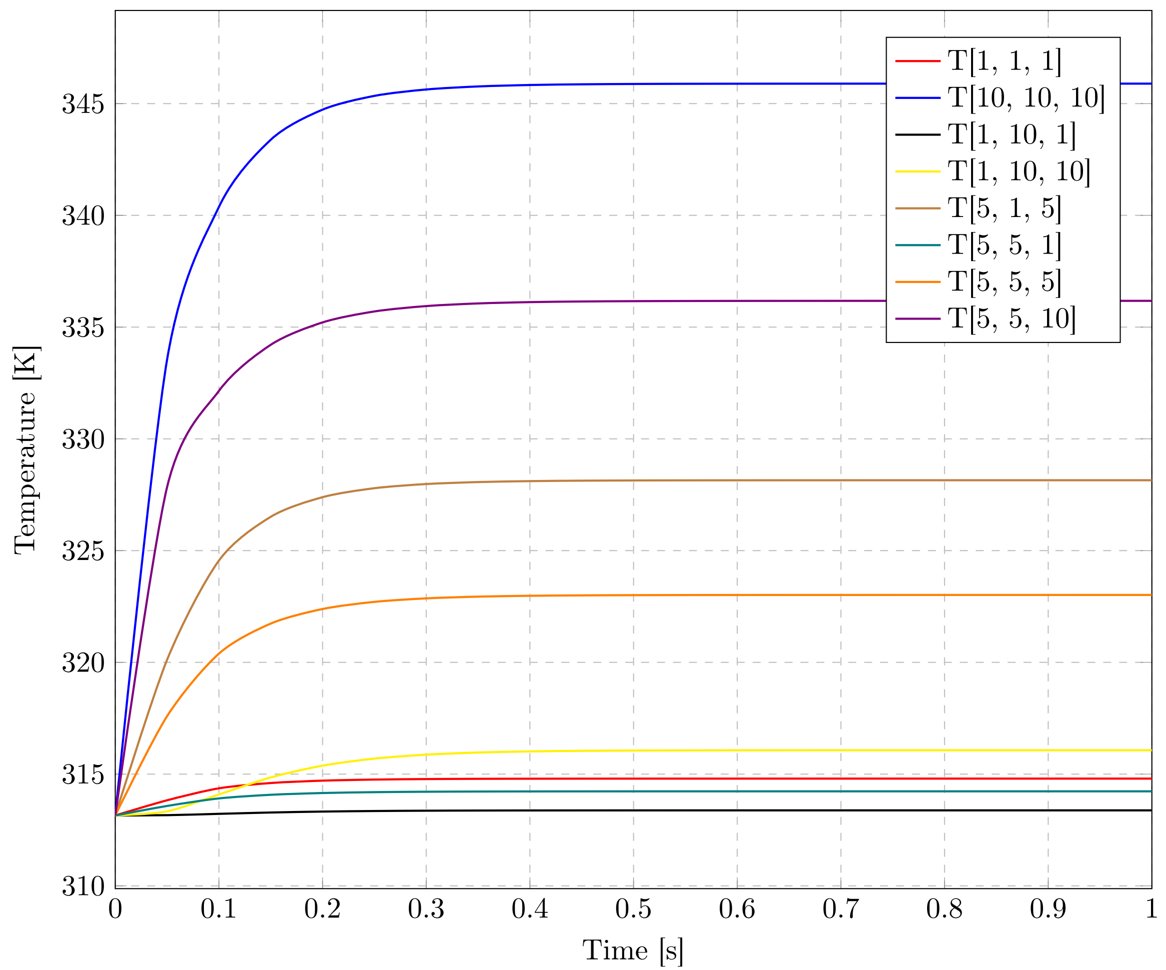


Figure 3.1: Behavior of the Thermal Chip with M = 10, N = 10, P = 10

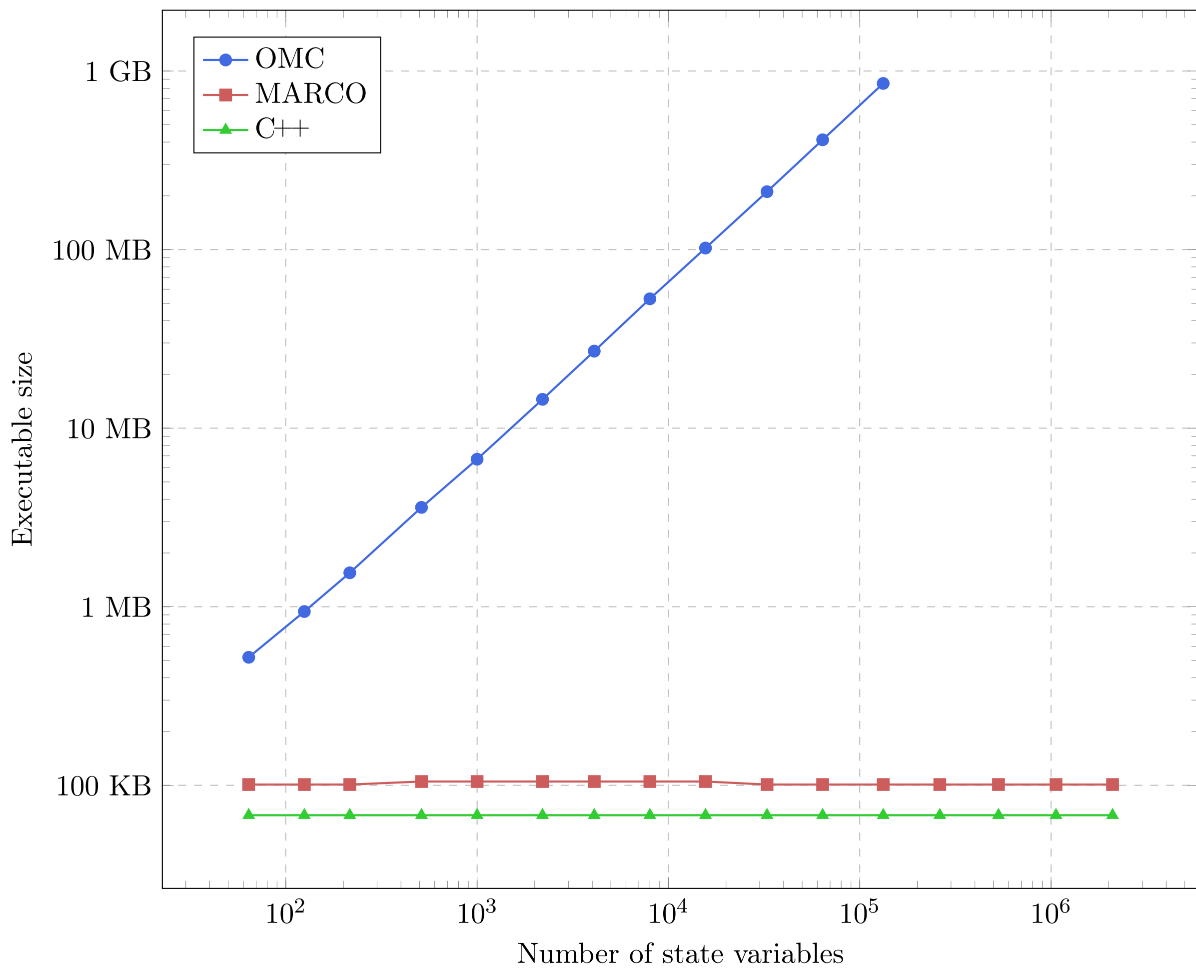


Figure 3.2: Binary size comparison

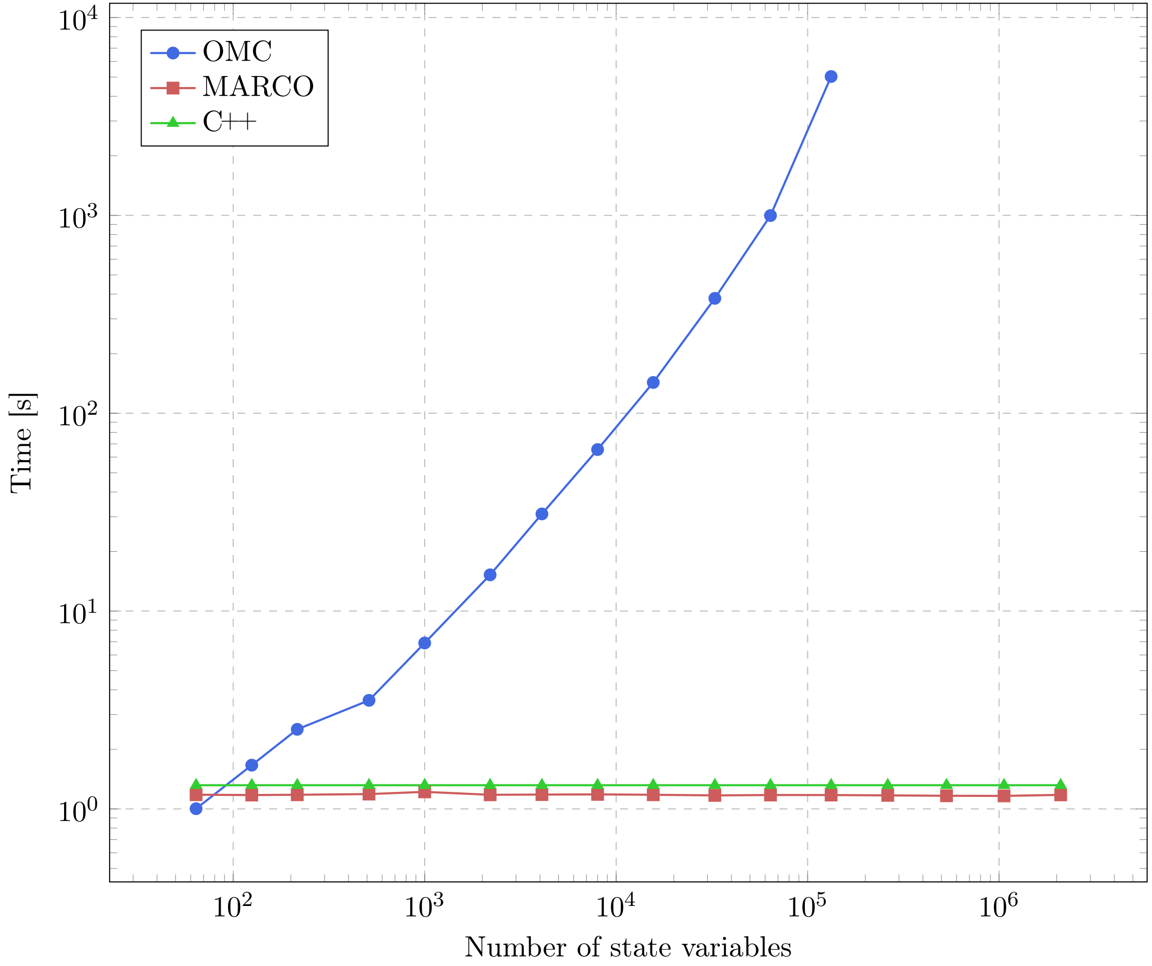


Figure 3.3: Compilation time comparison

3.4. Compilation Time

Figure 3.3 shows the times required to create the three simulation executables. It must be noted that the compilation time of MARCO also comprehends the OMC flattening process and the LLVM back-end execution. Again, it is clear that the objective of compiling the model in constant time with respect to the number of scalar equations has been reached. Both MARCO and C++ always require 1 to 2 seconds independently from the value of M, N and P. C++ compilation time does not present any fluctuations since it is only compiled once. The MARCO executable must be recompiled for every different model and therefore shows negligible differences. The compilation time of OMC, on the other hand, grows exponentially with respect to the number of scalar variables, reaching up to 5 000 seconds with 132 651 states. This is, again, caused by de-vectorization and loop unrolling.

3.5. Simulation Time

We discuss in this section the performance improvement of MARCO with respect to OMC for what concerns the simulation time, which is the time spent running the produced executables. The results are shown in figure 3.4. All three compilers produce simulations that scale quadratically with respect to the number of states. This is due to the behavior of the DAE solver that must compute several matrices while solving the non-trivial variables, such as the Jacobian matrix, which, even if sparse, can hypothetically reach the size $m \times m$, where m is the number of scalar variables computed by IDA. For this reason, which was confirmed through tests, almost the whole running time is spent inside calls to the IDA library. That said, we can notice that the executable produced by MARCO is around twice as fast as the one produced by OMC, thus still showing a significant advantage. Furthermore, the MARCO simulation time is practically the same as the C++ one, except for the first two iterations. For this reason, we can conclude that MARCO is able to obtain the optimal simulation of a model using a DAE solver, and the time bottleneck is in fact the execution of the IDA library.

In figure 3.5, the aggregate time required to compile and simulate the models is shown. From here, it can be noticed that MARCO and C++ provide a significant advantage over OMC, especially between 512 and 15 625 state variables. This is because OMC is constrained mainly by the compilation time during the first iterations, differently from MARCO and C++. But it is then constrained by the simulation time after the 8 000 states mark has been reached.

3.6. Memory Usage

Finally, we also wish to benchmark the amount of RAM required by the simulations produced. The results are shown in figure 3.6. As it can be noticed, for the first few iterations the RAM usage is almost constant. This is due to the stack segment, text segment and shared library code contained in memory being constant for MARCO and C++. For small simulations, in fact, the size of these segments is greater than the size of the data segment, which in Linux is utilized for storing both global variables and the heap. Then, the memory usage starts to grow quadratically with respect to the number of states, reaching several GB when dealing with hundreds of thousands of variables. The RAM usage grows even more quickly in the simulation produced by OMC because of poor memory management due to the usage of garbage collection, and because in this case also the stack and text segments grow in size when increasing the number of equations.

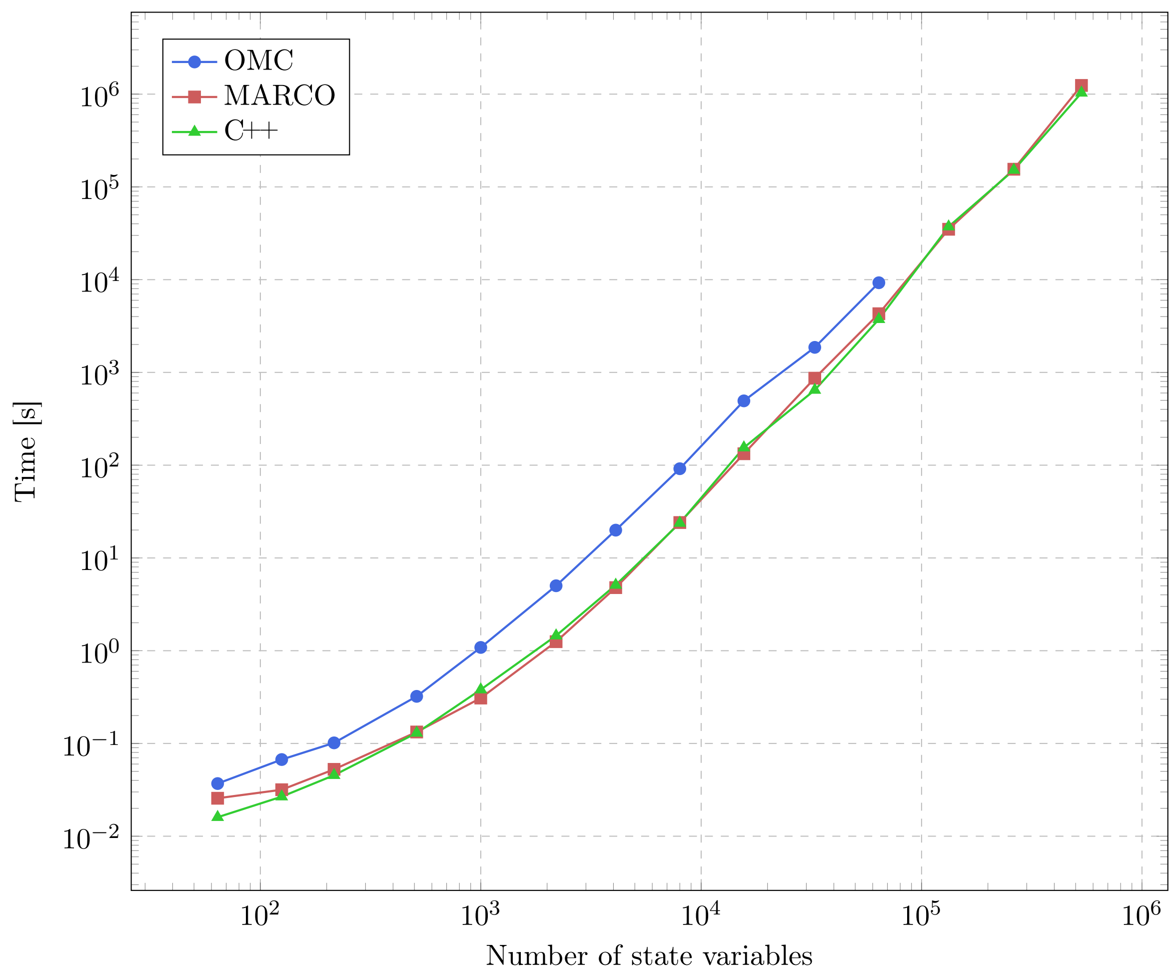


Figure 3.4: Simulation time comparison

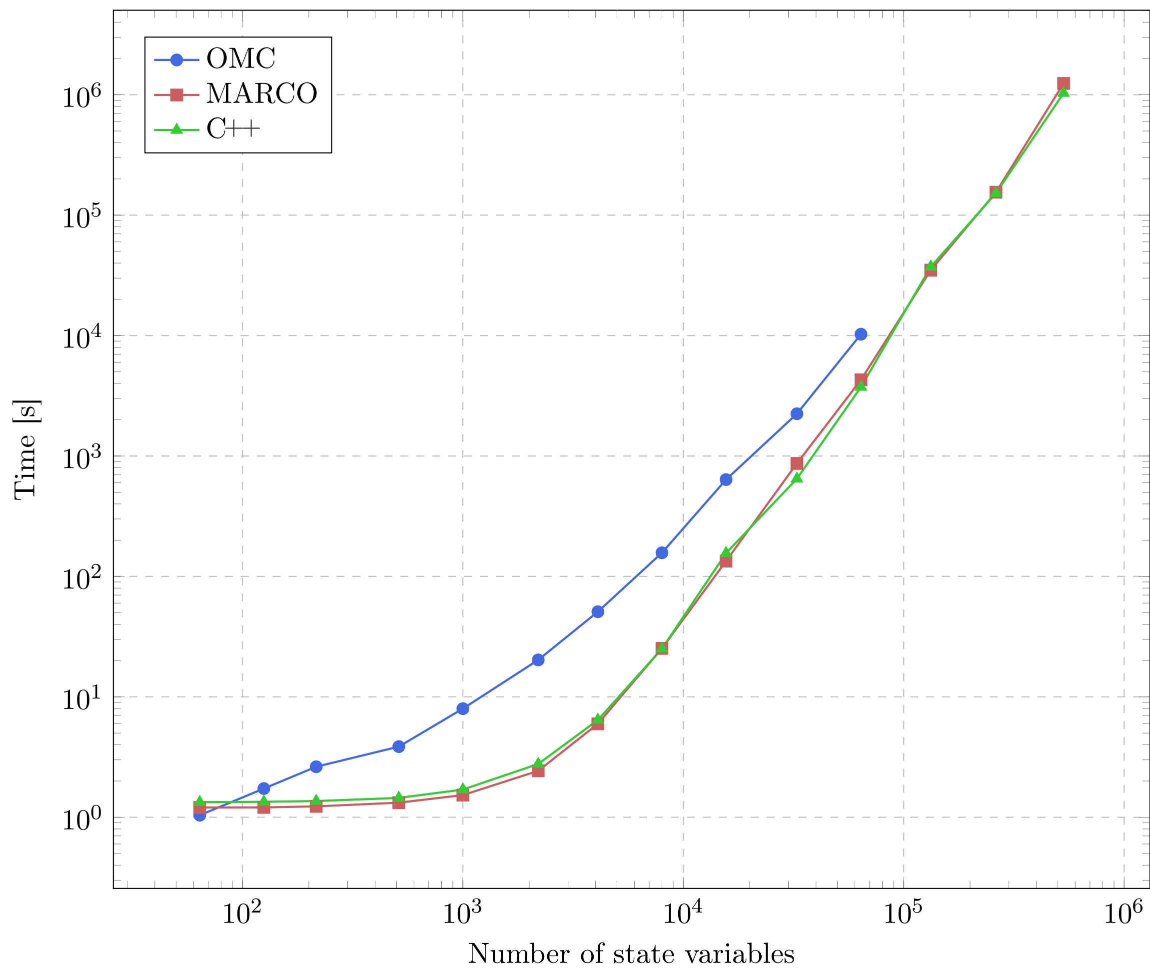


Figure 3.5: Compilation + simulation time comparison

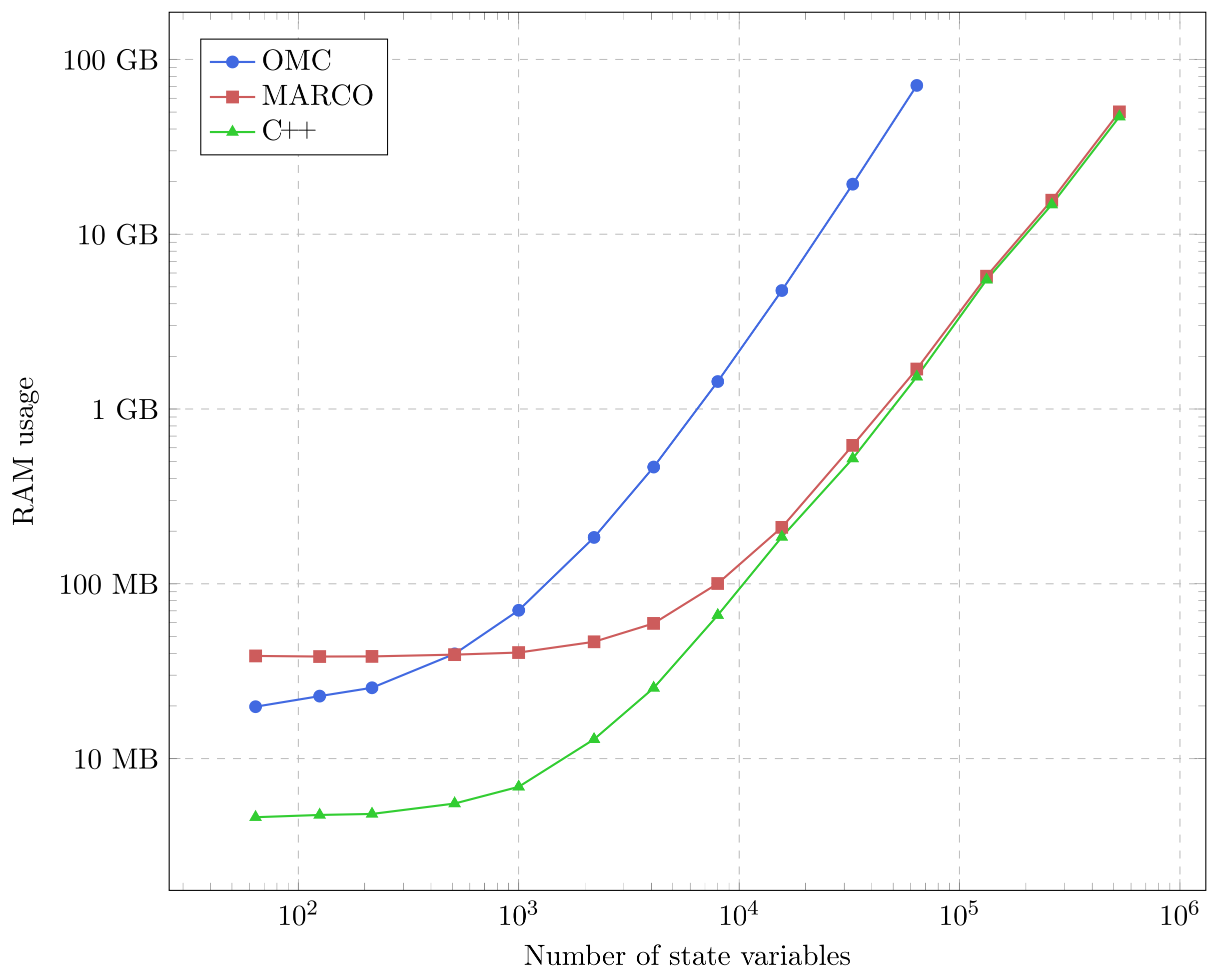


Figure 3.6: Memory usage comparison

4 | Conclusions

In this document we have presented the usefulness and the many difficulties of simulating real-world phenomena by means of differential and algebraic equations. Then we have shown a range of mathematical methods to overcome these problems and how these methods have already been implemented in market-ready compilers for the Modelica language. However, these compilers are not capable of effectively exploiting the design features of modern computer architectures, resulting in sub-par performance when compared to hand-written simulations or domain-specific tools. This is caused by de-vectorization of arrays and loop unrolling, which cause the binary executable to explode in size and make it impossible to exploit the locality properties of modern days computer architectures at runtime. For this reason, the newly born Modelica compiler called MARCO, currently with limited capabilities, has been introduced with the aim of overcoming this problem. The main objective of this new compiler is to preserve array structures and for loops inside the simulation executable. In this way, constant compilation time and efficient simulation code can be achieved.

Initially, the MARCO compiler only offered the Forward Euler method as a mean to solve differential equations. As it was shown, this method presents several restrictions on the types of models it can solve along with inefficiencies when dealing with stiff DAE systems. In this thesis we described the work needed to overcome these limitations, which consisted in introducing support for the variable step size DAE solver called IDA. All this was achieved while maintaining the original feature of preserving arrays and object-oriented structures as much as possible.

In particular, at first we discussed a simple "brute-force" or "raw" implementation of the DAE solver. Then, a more efficient design has been presented, which takes advantage of the sparsity of real-world phenomena and aims at using the DAE solver only for those variables that strictly requires it. It has been shown that such variables, which we call non-trivial variables, are the ones contained in algebraic loops, implicit equations and differential equations. For this reason, the MARCO pipeline has been modified to accommodate the need to identify such non-trivial variables. Then, a new transformation pass

has been introduced that takes all this information, extrapolates the equations that must be solved through IDA, creates the appropriate functions needed by IDA to compute the residual error and the Jacobian matrix of the system, and generates appropriate calls to the external IDA library. All of this was done through the introduction of a new MLIR dialect inside MARCO to facilitate the translation process, make it more maintainable and to make intermediate representations more readable.

Finally, the usefulness of this approach has been evaluated through several tests. First, the correctness of the produced results has been demonstrated. Then, it was shown that the original objective of creating a constant size binary executable in constant compilation time with respect to the number of scalar variables has been preserved. Finally, the improvement on the simulation with respect to the OpenModelica compiler, both in term of required time and memory, has been shown. These results have also been proven to be almost the same as hand-written C++ code, thus revealing that the only bottleneck of the simulation is given by the external DAE solver.

4.1. Future Works

The MARCO compiler can be further extended in many ways, both in term of parsing capabilities and solving methodologies. Here, we propose some of the possible future improvements.

Front-end support for records and ragged arrays As explained in section 1.7.1, the frontend of MARCO is currently unable to parse some of the data structures available in the Modelica standard. In particular, records are highly important structures of the Modelica language because they allow to describe components in an object-oriented fashion. Furthermore, supporting ragged multidimensional array inside MARCO is also of great importance since these often arise from the flattening of Modelica classes.

More efficient resolution of algebraic loops The Tiernan algorithm already present in MARCO and modified in this document is not the most efficient way of resolving strongly connected components inside a model. This method and the topological sorting inside the scheduling pass could be substituted with the more efficient Tarjan algorithm and Tearing method shown in section 1.2.1. This way, solving algebraic loops and scheduling the equations can be done in a single pass and with a better time complexity.

Introduction of Pantelides algorithm and Dummy Derivatives As explained in section 1.2.2, we assumed that we only operate on index-1 DAE systems since higher-

index models can be transformed into a corresponding index-1 system. This could be done inside MARCO through the implementation of the Pantelides algorithm and the Dummy Derivatives, so that the compiler will be able to handle such systems itself without the need of external tools to reduce the index of a model.

Implementation of other solvers With this thesis, the MARCO compiler now offers two ways for solving a model: the Forward Euler method and the DAE solver IDA. It is important to extend the range of available solvers to other methods, for example the Backward Euler or the Runge-Kutta algorithm, to give modelers the possibility of choosing the most appropriate solver for their models.

Complement explicit integration algorithms with an implicit solver As an alternative to using a single solver for both the differential and the algebraic equations, it is possible to solve only implicit systems with an implicit solver and use an explicit integration method to solve the differential equations. This way we would be able to solve implicit systems while still using an explicit integration method to exploit its better performance when dealing with non-stiff systems.

Filter away non requested variables Most of the time a modeler is only interested in the behavior of some of the variables contained in the model. For this reason, it is possible to filter the output so that it only shows the values of interest. If this information is known at compile time, it would be possible to remove completely those variables from the simulation, thus greatly reducing the required simulation time.

Bibliography

[1] G. Agosta, E. Baldino, F. Casella, S. Cherubin, A. Leva, and F. Terraneo. Toward a high-performance Modelica compiler. In *Proceedings of the 13th International Modelica Conference*, pages 313–320, Regensburg, Germany, March 2019. doi: 10.3384/ecp19157313.

[2] G. Agosta, F. Casella, S. Cherubin, A. Leva, and F. Terraneo. Towards a benchmark suite for high-performance Modelica compilers. In *9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, November 2019. doi: 10.1145/3365984.3365988.

[3] P. Aivaliotis, K. Georgoulias, Z. Arkouli, and S. Makris. Methodology for enabling Digital Twin using advanced physics-based modelling in predictive maintenance. *Procedia CIRP*, 81:417–422, June 2019. doi: 10.1016/j.procir.2019.03.072.

[4] E. Baldino. Structural pitfalls of state-of-the-art Modelica compilers: an explorative analysis. Master's thesis, Politecnico di Milano, 2018.

[5] W. Braun, F. Casella, and B. Bachmann. Solving large-scale Modelica models: New approaches and experimental results using OpenModelica. In *Proceedings of the 12th International Modelica Conference*, pages 557–563, Prague, Czech Republic, May 2017. doi: 10.3384/ecp17132557.

[6] K. E. Brenan, S. L. Campbell, and L. R. Petzold. Numerical solution of initial-value problems in differential-algebraic equations. In *Classics in Applied Mathematics*, Philadelphia, USA, 1996. SIAM. ISBN 978-0898713534.

[7] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal on Scientific Computing*, 15(6):1467–1488, November 1994. doi: 10.1137/0915088.

[8] D. Brück, H. Elmqvist, S. E. Mattsson, and H. Olsson. Dymola for multi-engineering modeling and simulation. In *Proceedings of the 2nd International Modelica Conference*, pages 55-1–55-8, Oberpfaffenhofen, Germany, March 2002.

[9] F. Casella. Simulation of large-scale models in Modelica: State of the art and future perspectives. In *Proceedings of the 11th International Modelica Conference*, pages 459–468, Versailles, France, September 2015. doi: 10.3384/ecp15118459.

[10] F. Casella and A. Leva. Modelling of thermo-hydraulic power generation processes using Modelica. *Mathematical and Computer Modelling of Dynamical Systems*, 12 (1):19–33, February 2006. doi: 10.1080/13873950500071082.

[11] F. Casella, M. Otter, K. Proelss, C. Richter, and H. Tummescheit. The Modelica fluid and media library for modeling of incompressible and compressible thermo-fluid pipe networks. In *Proceedings of the 5th International Modelica Conference*, pages 631–640, Vienna, Austria, September 2006.

[12] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, 2010. ISBN 978-1441938633.

[13] C. Cimino, E. Negri, and L. Fumagalli. Review of Digital Twin applications in manufacturing. *Computers in Industry*, 113, October 2019. doi: 10.1016/j.compind. 2019.103130.

[14] Dassault Systèmes. Dymola Homepage, 2022. URL `https://www.3ds.com/products-services/catia/products/dymola`.

[15] T. A. Davis and E. P. Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software*, 37(3): 36:1–36:17, September 2010. doi: 10.1145/1824801.1824814.

[16] P. J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. doi: 10.1145/1070838.1070856.

[17] H. Elmqvist, M. Otter, A. Neumayr, and G. Hippmann. Modia - equation based modeling and domain specific algorithms. In *Proceedings of the 14th International Modelica Conference*, pages 73–86, Linköping, Sweden, September 2021. doi: 10. 3384/ecp2118173.

[18] E. Fehlberg. Classical fifth-, sixth-, seventh-, and eighth-order Runge-Kutta formulas with stepsize control. NASA Technical Report, USA, 1968.

[19] M. Fioravanti. M.A.R.C.O.: an experimental high-performance Modelica compiler for large scale systems. Master's thesis, Politecnico di Milano, 2020.

[20] J. Frenkel, C. Schubert, G. Kunze, P. Fritzson, M. Sjölund, and A. Pop. Towards a benchmark suite for Modelica compilers: Large models. In *Proceedings of the 8th*

International Modelica Conference, pages 143–152, Dresden, Germany, March 2011. doi: 10.3384/ecp11063143.

[21] J. Frenkel, G. Kunze, and P. Fritzson. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Proceedings of the 9th International Modelica Conference*, pages 433–442, Munich, Germany, September 2012. doi: 10.3384/ecp12076433.

[22] P. Fritzson. *The Modelica Standard Library*, chapter 5, pages 157–168. Wiley - IEEE Press, September 2011. ISBN 978-1118010686. doi: 10.1002/9781118094259.ch5.

[23] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1588–1595, October 2005. doi: 10.1109/CACSD-CCA-ISIC.2006.4776878.

[24] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.

[25] A. C. Hindmarsh, R. Serban, C. J. Balos, D. J. Gardner, D. R. Reynolds, and C. S. Woodward. *User Documentation for IDA*. LLNL Computing, v5.7.0 edition, 2022.

[26] G. Kron. *Diakoptics: The Piecewise Solution of Large-Scale Systems*. Macdonald Publishing, London, United Kingdom, 1963.

[27] J. D. Lambert. *Numerical Methods for Ordinary Differential Systems: The Initial Value Problem*, chapter 6, pages 213–260. John Wiley, New York, USA, 1991. ISBN 978-0471929901.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, USA, March 2004. doi: 10.1109/CGO.2004.1281665.

[29] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *International Symposium on Code Generation and Optimization*, pages 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

[30] LLNL Computing. SUNDIALS, 2022. URL `https://computing.llnl.gov/projects/sundials`.

[31] LLVM Organization. LLVM Homepage, 2022. URL `https://llvm.org`.

[32] LLVM Organization. MLIR Homepage, 2022. URL `https://mlir.llvm.org`.

[33] S. E. Mattsson and G. Soderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993. doi: 10.1137/0914043.

[34] S. E. Mattsson, H. Elmqvist, and M. Otter. Physical system modeling with Modelica. *Control Engineering Practice*, 6(4):501–510, 1998. doi: 10.1016/S0967-0661(98)00047-1.

[35] R. Mejia-Gutierrez and R. Carvajal-Arango. Design verification through virtual prototyping techniques based on systems engineering. *Research in Engineering Design*, 28(4):477–494, February 2017. doi: 10.1007/s00163-016-0247-y.

[36] Modelica Association. *Modelica - A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.5*. Modelica Association, February 2021.

[37] Modelica Association. Modelica Homepage, 2022. URL `https://modelica.org`.

[38] M. Nikolic. A benchmarking framework for performance evaluation of Modelica compilers. Master's thesis, Politecnico di Milano, 2021.

[39] Open Source Modelica Consortium. OpenModelica Homepage, 2022. URL `https://www.openmodelica.org`.

[40] M. Otter and C. Schlegel. Symbolic generation of efficient simulation codes for robots. In *Proceedings Second European Simulation Muli-Conference*, pages 119–122, Nice, France, 1988.

[41] M. Otter, H. Elmqvist, and F. E. Cellier. Modeling of multibody systems with the object-oriented modeling language Dymola. *Nonlinear Dynamics*, 9(1):91–112, 1996. doi: 10.1007/BF01833295.

[42] M. Otter, H. Elmqvist, and F. E. Cellier. Relaxating - a symbolic sparse matrix method exploiting the model structure in generating efficient simulation code. In *Proceedings Symposium on Modeling, Analysis and Simulation, CESA '96, IMACS Multi-Conference on Computation Engineering in Systems Applications*, pages 1–12, Lille, France, 1996.

[43] C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing*, 9(2):213–231, 1988. doi: 10.1137/0909014.

[44] L. R. Petzold. A description of DASSL: a differential/algebraic system solver. In *IMACS World Congress*, Montreal, Canada, September 1982.

[45] A. Pop and P. Fritzson. MetaModelica: A unified equation-based semantical and mathematical modeling language. In *7th Joint Modular Languages Conference*, pages 211–229, September 2006. doi: 10.1007/11860990_14.

[46] M. Schluse, M. Priggemeyer, L. Atorf, and J. Romann. Experimentable Digital Twins - streamlining simulation-based systems engineering for industry 4.0. *IEEE Transactions on Industrial Informatics*, 14(4):1722–1731, April 2018. doi: 10.1109/TII.2018.2804917.

[47] N. Schröder, O. Lenord, and R. Lange. Enhanced motion control of a self-driving vehicle using Modelica, FMI and ROS. In *Proceedings of the 13th International Modelica Conference*, pages 441–450, Regensburg, Germany, March 2019. doi: 10.3384/ecp19157441.

[48] G. Schweiger, H. Nilsson, J.-P. Schöggl, W. Birk, and A. Posch. Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms. *Applied Mathematics and Computation*, 365, January 2020. doi: 10.1016/j.amc.2019.124713.

[49] M. Scuttari. Design and implementation of a Modelica compiler with MLIR and LLVM. Master's thesis, Politecnico di Milano, 2021.

[50] E. F. Sowell and P. Haves. Efficient solution strategies for building energy system simulation. *Energy and Buildings*, 33(4):309–317, April 2001. doi: 10.1016/S0378-7788(00)00113-4.

[51] F. Tao, M. Zhang, Y. Liu, and A. Nee. Digital Twin driven prognostics and health management for complex equipment. *CIRP Annals*, 67(1):169–172, May 2018. doi: 10.1016/j.cirp.2018.04.055.

[52] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi: 10.1137/0201010.

[53] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, December 1970. doi: 10.1145/362814.362819.

[54] M. Tiller. Modelica by example, 2014.

[55] J. van Amerongen and P. Breedveld. Modelling of physical systems for the design and control of mechatronic systems. *Annual Reviews in Control*, 27(1):87–117, December 2003. doi: 10.1016/S1367-5788(03)00010-5.

[56] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit. Modeling and optimization with Optimica and JModelica.org - languages and tools for solving large-scale dynamic optimization problems. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010. doi: 10.1016/j.compchemeng.2009.11.011.

A | ThermalChipDAE Benchmark

Listing A.1 Thermal chip DAE model used for benchmarking MARCO's DAE solver

```
package ThermalChipDAE
  package Types
    type Temperature = Real(unit = "K", nominal = 500);
    type Power = Real(unit = "W");
    type ThermalConductivity = Real(unit = "W/(m.K)");
    type ThermalConductance = Real(unit = "W/K");
    type SpecificHeatCapacity = Real(unit = "J/(kg.K)");
    type ThermalCapacitance = Real(unit = "J/K");
    type Density = Real(unit = "kg/m3");
    type Length = Real(unit = "m");
    type Time = Real(unit = "s");
  end Types;

  package Models
    partial model BaseThermalChip
      parameter Integer N = 4 "Number of volumes in the x direction";
      parameter Integer M = 4 "Number of volumes in the y direction";
      parameter Integer P = 4 "Number of volumes in the z direction";
      parameter Types.Length L = 12e-3 "Chip length in the x direction";
      parameter Types.Length W = 12e-3 "Chip width in the y direction";
      parameter Types.Length H = 4e-3 "Chip height in the z direction";
      parameter Types.ThermalConductivity lambda = 148 "Thermal
         conductivity of silicon";
      parameter Types.Density rho = 2329 "Density of silicon";
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity
          of silicon";
      parameter Types.Temperature Tstart = 273.15 + 40;
      final parameter Types.Length l = L / N "Chip length in the x
         direction";
      final parameter Types.Length w = W / M "Chip width in the y
         direction";
      final parameter Types.Length h = H / P "Chip height in the z
         direction";
```

```
  parameter Types.Temperature Tt = 273.15 + 40 "Prescribed temperature
     of the top surface";
  final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal
    capacitance of a volume";
  final parameter Types.ThermalConductance Gx = lambda*w*h / l "
    Thermal conductance of a volume, x direction";
  final parameter Types.ThermalConductance Gy = lambda*l*h / w "
    Thermal conductance of a volume, y direction";
  final parameter Types.ThermalConductance Gz = lambda*l*w / h "
    Thermal conductance of a volume, z direction";
  Types.Temperature T[N,M,P](each start = Tstart,each fixed = true) "
    Temperatures of the volumes";
  Types.Power Qx[N+1,M,P] "Power flows through the surfaces of the
    volumes, x direction";
  Types.Power Qy[N,M+1,P] "Power flows through the surfaces of the
    volumes, y direction";
  Types.Power Qz[N,M,P+1] "Power flows through the surfaces of the
    volumes, z direction";
  Types.Power Qb[N,M] "Power injected in the bottom volumes";
equation
  for i in 1:N loop
    for j in 1:M loop
      for k in 1:P loop
        C*der(T[i,j,k]) =  Qx[i,   j,k] + Qy[i,j,   k] + Qz[i,j,k   ]
                          -Qx[i+1,j,k] - Qy[i,j+1,k] - Qz[i,j,k+1] "
                             Energy balance on each volume";
      end for;
    end for;
  end for;
  for j in 1:M loop
    for k in 1:P loop
      Qx[1,j,k] = 0;
      for i in 2:N loop
        Qx[i,j,k] = Gx*(T[i-1,j,k] - T[i,j,k]);
      end for;
      Qx[N+1,j,k] = 0;
    end for;
  end for;
  for i in 1:N loop
    for k in 1:P loop
      Qy[i,1,k] = 0;
      for j in 2:M loop
        Qy[i,j,k] = Gy*(T[i,j-1,k] - T[i,j,k]);
      end for;
```

```
            Qy[i,M+1,k] = 0;
          end for;
        end for;
        for i in 1:N loop
          for j in 1:M loop
            Qz[i,j,1] = 2*Gz*(Tt - T[i,j,1]);
            for k in 2:P loop
              Qz[i,j,k] = Gz*(T[i,j,k-1] - T[i,j,k]);
            end for;
            Qz[i,j,P+1] = -Qb[i,j];
          end for;
        end for;
      end BaseThermalChip;

      model ThermalChipSimpleBoundary "Thermal chip model, constant power on
          half of the bottom surface"
        extends BaseThermalChip;
        parameter Types.Power Ptot = 100 "Total power consumption";
        final parameter Types.Power Pv = Ptot / (N * M / 2) "Power
            dissipated in a single volume";
      equation
        Qb[:, 1:div(M, 2)] = fill(Pv, N, div(M, 2));
        Qb[:, div(M, 2) + 1:end] = fill(0, N, M - div(M, 2));
      end ThermalChipSimpleBoundary;
    end Models;
  end ThermalChipDAE;
```

List of Figures

1.1 Electrical RLC Circuit [54] 6
1.2 Electrical RL Circuit [12] 8
1.3 SCCs found by the Tarjan algorithm in the RL electrical circuit 9
1.4 Electrical RC Circuit [12] 10
1.5 Three stage LLVM-based compiler pipeline 20
1.6 MARCO pipeline 22
1.7 Matching example 25
1.8 SCC example 26

2.1 DAG of the system example during the scheduling pass 34
2.2 MARCO pipeline with SUNDIALS IDA 38

3.1 Behavior of the Thermal Chip with M = 10, N = 10, P = 10 51
3.2 Binary size comparison 52
3.3 Compilation time comparison 53
3.4 Simulation time comparison 55
3.5 Compilation + simulation time comparison 56
3.6 Memory usage comparison 57

List of Tables

1.1 RLC electrical circuit incidence matrix 7
1.2 RL electrical circuit incidence matrix 9

2.1 DAE example 30
2.2 DAE example after SCC resolution 32
2.3 DAE example after the intermediate passes 33
2.4 DAE example after scheduling 34
2.5 DAE example after trivial variable substitution 37

3.1 Parameters used for the OMC, MARCO and C++ simulations 48
3.2 Absolute errors made by OMC, MARCO and C++ using tolerances equal to 10^{-6} against OMC using tolerances equal to 10^{-10} 50

List of Listings

1.1 C prototype of the user-defined IDA Residual function 12
1.2 C prototype of the user-defined approximated IDA Jacobian matrix 13
1.3 RLC circuit model [54] 14
1.4 Heat transfer in a one-dimensional rod [54] 15
1.5 Object-oriented RLC circuit model [54] 16
1.6 Algorithmic computation of a polynomial [54] 17
1.7 Heat transfer in a one-dimensional rod, flattened model 18
1.8 LLVM IR of the recursive factorial function 19
1.9 MLIR IR of the factorial function with a toy dialect 21
1.10 Matching example 25
1.11 SCC example 26
2.1 C++ prototype of the parametric Residual function of a vector equation . . 40
2.2 C prototype of the parametric Jacobian function of a vector equation . . . 42
A.1 Thermal chip DAE model used for benchmarking MARCO's DAE solver . 69

List of Algorithms

2.1 Trivial variable substitution 36
2.2 Residual function computation 41
2.3 Jacobian matrix computation 44
2.4 Main of the produced MARCO executable using IDA 46

List of Symbols

Variable	Description	SI unit
$\boldsymbol{f}$	Vector equation	-
$\boldsymbol{F}$	Residual function of a DAE system	-
$\hat{\boldsymbol{F}}$	Residual function of the non-trivial part of a DAE system	-
$\tilde{\boldsymbol{F}}$	Residual function of the trivial part of a DAE system	-
h	Time step of a simulation	s
$\boldsymbol{J}$	Jacobian matrix of a system	-
M	Number of volumes in the x direction	-
N	Number of volumes in the y direction	-
NEQ	Number of equations	-
NNZ	Number of non-zero values	-
P	Number of volumes in the z direction	-
$\boldsymbol{s}$	Trivial variables vector	-
t_0	Start time of a simulation	s
t_n	End time of a simulation	s
$\boldsymbol{u}$	Input variables vector	-
$\boldsymbol{v}$	Algebraic variables vector	-
$\boldsymbol{w}$	Non-trivial variables vector	-
$\boldsymbol{x}$	State variables vector	-
$\dot{\boldsymbol{x}}$	Derivative variables vector	-
$\boldsymbol{y}$	Unknown variables vector	-
$\boldsymbol{z}$	Known variables vector	-
t	Scalar time variable	s

Acknowledgments

This document represents the end of a growth path of five and a half years, which would not have been possible without the help and support of many people.

I would like first of all to thank my advisor Prof. Giovanni Agosta, together with all the co-advisors and the Modelica Working Group, for their guidance throughout this project.

My friends Alessio, Clarence, Federico, Francesco, Silvio and other friends also deserve my thanks. All of them, directly and indirectly, have provided me with insights and valuable suggestions during all my years of university.

Last but not least, I am also grateful to my family, whose constant love and support has kept me motivated and confident. My accomplishments and my successes are because they have always believed in me.