

POLITECNICO DI MILANO
DEPARTMENT OF ELECTRONICS, INFORMATION AND BIOENGINEERING
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

ADVANCING MODELICA WITH MLIR: ARRAY-AWARE COMPILATION AND SIMULATION

Doctoral Dissertation of
Michele Scuttari

Advisor:
Prof. Giovanni Agosta

Tutors:
Prof. Luciano Baresi
Prof. Luca Mottola

The Chair of the Doctoral Program:
Prof. Luigi Piroddi

A.Y. 2024 / 2025 – XXXVIII Cycle

A Mamma e Papà
che mi hanno sempre sostenuto

Abstract

MODELING languages are becoming increasingly important in scientific and engineering domains, and they are driving the need for advanced compilers that automatically translate their descriptive semantics into efficient simulations. As models grow in complexity and scale, state-of-the-art compilers often encounter significant performance challenges, particularly when handling array-oriented models. These limitations hinder the adoption of modeling languages in high-performance applications and therefore restrict the expressiveness and flexibility available to domain experts.

This thesis addresses these challenges by presenting novel algorithms, data structures, and compiler infrastructure specifically designed for the compilation and simulation of large-scale array-oriented models written in the Modelica language. The work makes three principal contributions that collectively overcome the scalability limitations of existing compilation approaches.

The first contribution introduces compact data structures for representing arbitrarily high-dimensional index spaces, together with efficient algorithms for their manipulation during compilation. These structures maintain constant-time or logarithmic-time complexity for common operations when the number of scalar array elements – and thus equations – grows. Building upon these foundations, the thesis presents array-aware algorithms for incidence matrix construction, matching, strongly connected component detection, and equation scheduling that avoid the prohibitive costs of traditional scalar expansion approaches. These algorithms form the backbone of an array-aware causalization pipeline that processes models at the array level rather than decomposing them into individual scalar elements, whenever possible.

The second contribution addresses the need for modern compiler infrastructure capable of preserving high-level semantic information throughout the compilation process. The thesis presents a multi-level intermediate representation based on the MLIR framework, introducing custom dialects that capture differential and algebraic equation semantics, numerical solver interfaces, and runtime system operations. This modular architecture enables clear separation of concerns, with each dialect addressing distinct aspects of the compilation pipeline while maintaining seamless integration with LLVM's mature optimization and code generation infrastructure.

The third contribution implements a multithreaded runtime environment engineered to exploit parallelism on modern multi-core processors. The compiler identifies independent equations and generates code that the runtime system distributes across available processor cores, translating compiler-identified parallelism into concrete performance improvements during simulation execution.

Experimental evaluation against the industry-grade OpenModelica compiler demonstrates the effectiveness of these contributions. The experimental results show that MARCO achieves near-constant compilation times as model size increases, contrasting sharply with traditional approaches that exhibit linear or worse scaling behavior. For instance, in the most complex examined case of object-oriented modeling of a silicon chip with cooling channels, MARCO compiled models with over 30 million scalar equations in approximately 2 seconds – the same time required for much smaller versions of the same model. In contrast, OpenModelica exhibited linear scaling, requiring up to ~ 30 minutes of compilation time for models with $\sim 300\,000$ equations and being unable to handle models of larger size with the same hardware resources. Beyond compilation efficiency, simulation experiments demonstrated that the runtime environment efficiently exploits modern multi-core processors. As model size increases, MARCO's superior scalability becomes evident: for the same aforementioned thermal model, MARCO achieves simulation speedups ranging from $4.5\times$ to $9\times$ compared to OpenModelica, depending on the employed numerical integration method, but also up to two orders of magnitude for the simpler thermal version without cooling channels.

These results demonstrate that the combination of array-aware data structures and algorithms, modern multi-level compiler infrastructure, and parallel runtime support fundamentally transforms the scalability characteristics of differential and algebraic equation compilation. The presented solutions enable the practical simulation of large-scale cyber-physical systems that were previously computationally infeasible, paving the way for broader adoption of declarative modeling languages in high-performance computational science and engineering applications.

Sommario

I linguaggi di modellazione stanno diventando sempre più importanti nei domini scientifici e ingegneristici, e stanno alimentando la necessità di compilatori che traducano automaticamente la loro semantica descrittiva in simulazioni efficienti. Tuttavia, con l'aumentare della complessità e della scala dei modelli, le attuali soluzioni spesso incontrano significative sfide prestazionali, in particolare nella gestione di modelli basati su semantiche vettoriali. Queste limitazioni ostacolano l'adozione dei linguaggi di modellazione e quindi restringono l'espressività e la flessibilità a disposizione degli esperti.

Questa tesi affronta queste sfide presentando algoritmi, strutture dati e infrastrutture di compilazione innovative, specificamente progettati per la compilazione e simulazione di modelli orientati ai vettori su larga scala scritti nel linguaggio Modelica. Il lavoro fornisce tre contributi principali che collettivamente superano le limitazioni di scalabilità degli approcci di compilazione esistenti.

Il primo contributo introduce strutture dati compatte per rappresentare spazi di indici di dimensionalità arbitrariamente elevata, insieme ad algoritmi efficienti per la loro manipolazione durante la compilazione. Queste strutture mantengono complessità temporale costante o logaritmica per le operazioni comuni quando il numero di elementi scalari dei vettori – e quindi le equazioni – aumenta. Basandosi su queste fondamenta, la tesi presenta algoritmi per la costruzione della matrice di incidenza, l'associazione variabile-equazione, il rilevamento delle componenti fortemente connesse e l'ordinamento delle equazioni che evitano i costi proibitivi degli approcci tradizionali basati sull'espansione scalare. Questi algoritmi formano la spina dorsale di una pipeline di causalizzazione che processa i modelli a livello di vettori piuttosto che decomporli in singoli elementi scalari, quando possibile.

Il secondo contributo affronta la necessità di un'infrastruttura di compilazione moderna capace di preservare informazioni semantiche di alto livello durante tutto il processo di compilazione. La tesi presenta una rappresentazione intermedia multi-livello basata sul framework MLIR, introducendo dialetti personalizzati che catturano la semantica di equazioni differenziali e algebriche, le interfacce dei solutori numerici e le interfacce dell'ambiente di simulazione. Questa architettura modulare consente una chiara separazione delle responsabilità, con ciascun dialetto che affronta aspetti dis-

tinti del processo di compilazione mantenendo al contempo un'ottima integrazione con l'infrastruttura di ottimizzazione e generazione di codice di LLVM.

Il terzo contributo implementa un ambiente di simulazione progettato per sfruttare il parallelismo sui moderni processori multi-core. Il compilatore identifica le equazioni indipendenti e genera codice che il sistema di runtime distribuisce sulle risorse hardware disponibili, traducendo il parallelismo identificato dal compilatore in miglioramenti concreti delle prestazioni durante l'esecuzione della simulazione.

La valutazione sperimentale confrontata con il compilatore OpenModelica dimostra l'efficacia di questi contributi. I risultati sperimentali mostrano che MARCO raggiunge tempi di compilazione costanti all'aumentare delle dimensioni del modello, in netto contrasto con gli approcci tradizionali che mostrano comportamenti lineari o peggiori. Per esempio, nel caso più complesso esaminato di modellazione orientata agli oggetti di un chip di silicio con canali di raffreddamento, MARCO ha compilato modelli con oltre 30 milioni di equazioni scalari in circa 2 secondi — lo stesso tempo richiesto per versioni molto più piccole dello stesso modello. Al contrario, OpenModelica ha mostrato un incremento lineare, richiedendo fino a ~ 30 minuti di tempo di compilazione per modelli con $\sim 300\,000$ equazioni e risultando incapace di gestire modelli di dimensioni maggiori con le stesse risorse hardware. Oltre all'efficienza di compilazione, gli esperimenti di simulazione hanno dimostrato che l'ambiente di runtime sfrutta efficientemente i moderni processori multi-core. All'aumentare delle dimensioni del modello, la superiore scalabilità di MARCO diventa evidente: per lo stesso modello termico sopra menzionato, MARCO raggiunge incrementi prestazionali che vanno da $4.5\times$ a $9\times$ rispetto a OpenModelica, a seconda del metodo di integrazione numerica impiegato, ma anche fino a due ordini di grandezza per il modello termico più semplice senza canali di raffreddamento.

Questi risultati dimostrano che la combinazione di strutture dati e algoritmi vettoriali, infrastruttura di compilazione moderna multi-livello e ambiente di runtime parallelo trasforma fondamentalmente le caratteristiche di scalabilità della compilazione di equazioni differenziali e algebriche. Le soluzioni presentate abilitano la simulazione pratica di sistemi ciber-fisici su larga scala che erano precedentemente computazionalmente infattibili, aprendo la strada a una più ampia adozione dei linguaggi di modellazione dichiarativi nelle applicazioni di scienza computazionale e ingegneria ad alte prestazioni.

Acknowledgements

This thesis would not have been possible without the support and contribution of many people and organizations.

I am deeply grateful to my parents for their unconditional love, continuous encouragement, and unwavering belief in me throughout this journey.

I would like to thank my friends for the joyful moments they brought during this years. Special thanks go to **Luca Cuzzocrea**, with whom I shared countless interests and conversations which enriched my professional knowledge, and to **Daniele Perossa**, with whom, even if in different departments, I shared the PhD timeline and associated challenges.

I would like to express my gratitude to my supervisor, Professor **Giampaolo Agosta**, who gave me the freedom to explore and develop my own ideas while at the same time providing guidance that was always on point, as well as for his support through difficult times.

I am also grateful to the rest of the university team, which continuously participated in the development process, of which I would like to especially thank Professor **Francesco Casella** for the many explanations and suggestions regarding the modeling and simulation domains.

I would like to thank Professor **Stefano Cherubin** for his availability in making the six-months visiting period in Trondheim possible, and nonetheless for the fun time we had in improving our competitive sport skills.

I am grateful to **Politecnico di Milano** and **Huawei Pisa Research Center** for the financial support that made this journey possible. This research received funding from the Italian Ministry of University and Research (MIUR) within the National Recovery and Resilience Plan (PNRR), and from the European Union (EU) as part of the NextGenerationEU project.

Last but not least, I would like to thank **Paolo Belluco**, **Samuele Polistina**, **Andrea Belluco**, and the whole team of **LWT^3** for hosting me during my third-year internship, as well as for the invaluable time spent together and personal advices for which they were always available.

Contents

Acknowledgements **V**

1 Introduction **1**
1.1 The Importance of DAE-Based Modeling Languages 1
1.2 The Challenges of Compiling and Simulating Large-Scale Models . . . 2
1.3 Research Questions 2
1.4 Contributions 3
1.5 Dissertation Structure 4

2 The Modeling and Simulation Landscape **5**
2.1 Domains and Classification 5
2.2 The Modeler, the Simulator, and the Simulation 7
2.3 Foundations of Continuous Dynamic Systems 8
2.4 Software Solutions for Continuous Systems 11
2.5 Modelica and Base Modelica 13
2.6 Limitations of State of the Art Modelica Compilers 15

3 Array-Aware Compilation **21**
3.1 Multidimensional Space Indexing 21
3.2 Mapping Equation to Variable Indices 28
3.3 Array-Aware Causalization 33
3.3.1 Data Structures for Array-Aware Matching 33
3.3.2 Computation of Strongly-Connected Components 36
3.3.3 Equations Sorting 39

4 MLIR Dialects for DAE Systems **43**
4.1 A Multi-Level Approach to Compilation 43
4.2 Bringing MLIR to the Modelica Environment 45
4.3 Base Modelica Dialect 46
4.4 Solver Dialects 52
4.5 Runtime Dialect 56

5 MLIR Compilation Pipeline **57**
5.1 Canonicalization . . . 58
5.1.1 Automatic Differentiation . . . 58
5.1.2 Derivatives Materialization . . . 59
5.2 Model Solving . . . 63
5.2.1 Matching . . . 64
5.2.2 Variables Pruning . . . 66
5.2.3 SCCs Computation . . . 66
5.2.4 Numerical Integration . . . 69
5.2.5 Scheduling . . . 72
5.2.6 Closed-form Solution of Equations . . . 72
5.2.7 Iterative Solution of Implicit Equations and SCCs . . . 73
5.2.8 Schedule Parallelization . . . 75
5.3 Dialects Lowering . . . 76
5.3.1 Conversion of Equation Schedules . . . 77

6 Runtime Environment **81**
6.1 Conceptual Overview . . . 81
6.2 Driver Libraries . . . 82
6.2.1 Main Driver and Simulation Workflow . . . 82
6.2.2 Explicit Euler . . . 84
6.2.3 IDA . . . 86
6.3 Utility Libraries . . . 87
6.4 Integration Libraries . . . 89
6.5 Language Support Libraries . . . 89

7 Evaluation **91**
7.1 Qualitative evaluation of compiler design . . . 91
7.2 Quantitative evaluation of compiler performance . . . 92

8 Conclusions **105**
8.1 Answer to Research Questions . . . 105
8.2 Performance Achievements . . . 107
8.3 Future Works . . . 107

Bibliography **109**

A Benchmarks **115**
A.1 Wire . . . 115
A.2 Silicon Chip: ODE . . . 115
A.3 Silicon Chip: DAE . . . 118
A.4 Silicon Chip: OO . . . 120
A.5 Silicon Chip with Cooling Channels . . . 122
A.6 Heat Exchangers . . . 124

CHAPTER 1

Introduction

MANY engineering domains [75, 60, 52], mechanical, electrical, and chemical engineering, to name some of the main ones, heavily rely on dynamic modeling and simulation for the design and operation of complex systems. For example, electrical grids [30, 72] are becoming *smart* to account for the variability of renewable energy sources and the increased demand introduced by electric vehicles. Similarly, industrial production is being revolutionized by Industry 4.0 technologies, including digital twins [48, 10, 46], which are heavily based on simulation. As a result, the size and complexity of such models are rising rapidly, and require a rethink of the tools and technologies involved.

1.1 The Importance of DAE-Based Modeling Languages

At the heart of many simulation workflows lie Differential-Algebraic Equations (DAEs), which provide a natural mathematical representation for continuous-time systems described by physical laws. Unlike procedural programming languages where the developer must explicitly specify the solution algorithm, DAE-based modeling languages such as Modelica [53] allow engineers to describe *what* the system is rather than *how* to solve it. These languages offer constructs that, when combined, define a system of DAEs, rather than a procedural description of how to perform its numerical integration. Thus, they belong to the class of declarative languages. Even in languages where parts of the model can be expressed procedurally, procedural code is typically relegated to modeling parts of the system that are inherently procedural, such as control algorithms.

This declarative paradigm brings significant advantages: it enhances model readability, facilitates the reuse of component libraries, and allows domain experts to focus on the modeling aspect rather than the implementation of relatively low-level code.

However, this also introduces unique challenges. The compilation of DAE-based modeling languages is very different from typical programming languages and involves algorithms spanning multiple abstraction levels and domains, from mathematics to computer science.

1.2 The Challenges of Compiling and Simulating Large-Scale Models

The importance of this field has grown dramatically in recent years. As systems become larger and more complex – think of entire power grids, building energy systems, or multi-physics industrial processes – models can easily involve tens of millions of equations and variables. Many of these variables naturally represent multidimensional arrays, such as temperature distributions in spatial grids, pressure fields in fluid networks, or state vectors in large-scale control systems.

However, existing compilers often struggle to handle such array-oriented models efficiently [5]. From an internal representation perspective, most compilers rely on scalar expansion, where each array element is treated as an independent variable. While conceptually simple, this approach becomes prohibitively expensive for large-scale models: a three-dimensional array of size $100 \times 100 \times 100$ expands into one million scalar variables, discarding the high-level array semantics and any opportunity that may come from the recognition of structured access patterns.

From an algorithmic standpoint, traditional compilation techniques are, for the majority, also not designed with array awareness in mind. For example, matching algorithms that assign DAEs to variables must handle incidence matrices whose size grows quadratically with the number of scalar elements, leading to severe scalability issues in terms of compilation time and generated binary size. The absence of array-aware data structures and algorithms thus represents a critical bottleneck in the compilation of large-scale DAE-based models, limiting both the size of models that can be practically compiled and the quality of optimizations that can be applied.

Moreover, the simulation of such models requires modern approaches to leverage the capabilities of modern hardware architectures, such as multi-core processors and accelerators. To fully exploit these resources, both the compilation process and the runtime environment must be designed to support parallelism and efficient resource management. This involves developing compiler techniques that can analyze model structure, identify independent computations, and generate code optimized for concurrent execution, as well as implementing an execution logic capable of distributing workloads across the available hardware.

1.3 Research Questions

Despite significant advances in DAE-based modeling and simulation, several fundamental challenges thus remain unresolved when dealing with large-scale array-oriented models. This thesis addresses these challenges by investigating the following research questions.

RQ1: How can multidimensional arrays be efficiently represented and manipulated during the compilation of DAE-based modeling languages?

Current state-of-the-art compilers struggle with array-aware analyses and transformations. Scalar expansion of arrays leads to prohibitive memory consumption and compilation time for large models. There is a need for compact data structures that can represent high-dimensional index spaces and access patterns, enabling efficient array-aware causalization of the DAE system and code generation.

RQ2: Which changes within the compiler domain are required to effectively apply array-aware algorithms and improve the quality of code generation?

Traditional Modelica compilers lack the infrastructure to preserve high-level semantic information throughout the compilation pipeline, making it difficult to integrate array-aware data structures and algorithms. Research is needed to design and implement a modern compiler whose internal representations are able to preserve high-level semantics and progressively transform them into optimized executable code.

RQ3: How can the execution of DAE-based simulations be optimized to exploit parallel hardware architectures?

While compiler techniques can identify independent equations and operations, the actual performance gain depends on an efficient runtime environment. Research is needed to design and implement multithreaded runtime systems that can effectively schedule and execute independent computations, manage data dependencies, and integrate seamlessly with the compiler-generated code.

1.4 Contributions

Addressing the research questions outlined in Section 1.3, this doctoral thesis focuses on the challenges posed by large-scale array-oriented models by presenting novel techniques and internal compiler representations that advance the state of the art in both the model compilation and simulation execution domains.

To address Research Question 1 (RQ1), the thesis presents data structures that compactly represent points in arbitrarily high-dimensional spaces, together with algorithms for their efficient manipulation. These structures form the foundation for the whole compiler analyses and transformations. In particular, special kinds of access patterns are accelerated by leveraging such compact representations, and a new implementation for a multidimensional incidence matrix is proposed to improve the efficiency of state-of-the-art algorithms for array-aware matching. Novel algorithms for the detection of Strongly Connected Components (SCCs) and their ordering are also introduced to complete the implementation of an array-aware causalization pipeline.

To address Research Question 2 (RQ2), the thesis describes how to bring modern multi-level compilation into the Modelica ecosystem, introducing a set of MLIR dialects aiming to represent and progressively translate DAE systems that are initially described through the Base Modelica language. The introduction of these dialects responds to the need for preserving high-level semantics and building a modular and extensible representation that aligns with modern approaches to compilation.

Finally, to address Research Question 3 (RQ3), the array-aware analyses and representations within the compiler are complemented by the implementation of a runtime environment capable of multithreaded execution of independent equations. Such implementation allows for the exploitation of modern hardware architectures, where multiple computing units are available.

1.5 Dissertation Structure

The remainder of this thesis is organized as follows:

- Chapter 2 establishes the context of this work, surveys existing software for continuous system simulation, and analyzes the limitations of current Modelica compilers.
- Chapter 3 introduces novel data structures and algorithms for efficiently representing and manipulating high-dimensional points, and extends the state of the art with respect to array-aware compilation of DAE systems.
- Chapter 4 details the design of MLIR dialects for representing and progressively compiling DAE models written in the Base Modelica language.
- Chapter 5 describes key transformations of the compilation pipeline, illustrating the progressive lowering of descriptive Base Modelica semantics into an executable form.
- Chapter 6 presents the implementation of a multithreaded runtime environment and discusses its integration with the compiler.
- Chapter 7 evaluates the proposed solution on a set of array-oriented benchmarks, focusing on performance and scalability of both the compilation and simulation processes.
- Chapter 8 concludes the thesis and outlines directions for future research.

CHAPTER 2

The Modeling and Simulation Landscape

THIS chapter presents the variegated landscape of modeling and simulation. First, a theoretical background is provided on how models and simulations should be considered in the context of this work, both from a high-level perspective and from a more mathematical one. The focus is then narrowed to the simulation of continuous dynamic systems, and more specifically to the Modelica modeling language. The approaches and shortcomings of the existing tools supporting Modelica are analyzed, thus motivating the research line pursued in this thesis.

2.1 Domains and Classification

The concepts of modeling and simulation span across a wide range of domains, from engineering and architecture to science and entertainment. In civil construction, for instance, architects and engineers frequently employ scale models of buildings or infrastructure projects to showcase their designs to stakeholders. Similarly, the aerospace industry has long relied on experimental techniques during the design process, like wind tunnels to evaluate fluid dynamics and the forces acting on the aircrafts.

However, with the advent of the digital era, these processes have undergone a significant transformation. Models and simulations can now be represented and carried out in a digital manner, opening up new possibilities for innovation and efficiency. Following the previous examples, software packages have been developed to produce detailed two or three-dimensional renderings, which are essential to the emerging fields of augmented and virtual reality [27]. Computational Fluid Dynamics (CFD) has permeated the aerospace and aeronautic environments and is extensively used in conjunction with traditional experimental studies [50]. Flight simulators have become a well-established technology used in the training programs for current and future pilots [8, 73], enabling

detailed testing of dangerous scenarios and thus improving their reaction capabilities. Simulators have recently become an essential tool also in motorsport, enabling drivers to continuously improve their skill, and teams to improve both the performance of their racing vehicles [38, 47, 69], and the strategies used during competitions [11, 34].

Given the variety of possible interpretations, it becomes necessary to establish a common ground for what a model and a simulation are expected to be in the context of this manuscript. Here, a *model* is considered to be the simplified description of the *system* being investigated. The term *system* is used to indicate a cyber-physical system that can be described as the union of multiple, smaller, and simpler components. Moreover, we assume systems to be *dynamic systems*, meaning that their state is characterized by changes over time. These changes are the result of the so-called *behavior* of the system, and a *simulation* is the computation of this behavior within a virtual space-time environment.

The concept of *model* can be further refined according to the objectives it is designed to fulfill. Indeed, models serve diverse purposes, ranging from design and early evaluation to real-world integration and safety-critical deployments. For instance, a flight simulator exhibits a significantly different level of interaction with the real world compared to an object depicted in augmented or virtual reality. Some components of flight simulators may even integrate into production environments to enhance the safety of the aircraft. These different levels of interaction are represented by the concepts of *digital model*, *digital shadow*, and *digital twin* [40, 68].

A *digital model* refers to a virtual representation of a physical system that is primarily used to evaluate performance and characteristics under specific conditions without the need for real-time data integration. A *digital shadow* is a more advanced form of a digital model that instead incorporates real-time feeds from a physical system [1], which are used to reflect the current state of the system in the digital domain. This capability makes digital shadows useful for monitoring, anomaly detection, and informed decision-making. A *digital twin* further extends the functionality of a digital shadow by introducing bidirectional communication. This advancement allows the system to not only simulate scenarios and predict future states, but also to automate responses to specific conditions [64, 22].

The classification of models can also be based on the behavior of the systems they represent. *Continuous systems*, often encountered in the modeling of physical phenomena, are characterized by smooth changes, with no discontinuities. *Discrete systems*, by contrast, are characterized by changes occurring at precise points in time, and are often accompanied by discontinuities between the states of their components. It is important to note that the term *discrete* does not imply a quantization of the simulated time, nor of other dimensions. Indeed, a system with continuous components can be simulated using a discretized simulated time, incremented only by fixed amounts. Similarly, a discrete system can be simulated using a continuously-modeled time, which is therefore capable of assuming any value within the domain of real numbers. The two behaviors – continuous and discrete – are also not mutually exclusive [70]. *Combined systems* – also referred to as *hybrid systems* – incorporate both continuous and discrete behaviors [66, 44]. In these systems, continuous changes occur as long as the conditions expressed by the discrete conditions remain valid [18].

Another dimension for classifying models is the nature of their reproducibility. *De-*

terministic systems are entirely predictable; given a specific set of inputs, they always produce the same outputs. In contrast, *stochastic systems* incorporate probabilistic elements; their behaviors cannot be uniquely predicted, even with identical inputs, as randomness influences their outcomes.

2.2 The Modeler, the Simulator, and the Simulation

The variety of possible classifications – based on objectives, system behavior, and reproducibility – demonstrates how models can address a wide range of simulation needs. For this reason, creating and using software models, thus translating abstract concepts into actual simulations, requires a structured approach.

The process of implementing a simulation begins with identifying the problem or phenomenon to be studied. During this initial stage, it is crucial to clearly define the simulation's objectives, the questions it aims to answer, the outcomes to be predicted, and the scenarios to be tested.

The next step involves creating a high-level representation of the system. The expertise of the *modeler* should result in a first description – also called *conceptual model* [14] – that captures the essential components of the system, together with their interactions and the physical dimensions to be observed. Any suitable language or representation framework may be used for this purpose. For example, a continuous system may be represented as a set of equations, while the behavior of a discrete system may consist of a state diagram. Striking the right balance between complexity and accuracy during this phase is critical, as both depend on the simulation's objectives and the available computational resources.

The conceptual model is then transformed into a software program – also referred to as the *simulator* [76] – which embodies the behavior of the system. When executed, this software performs the intended *simulation*. At this stage, *variables* serve as the point of contact between the conceptual model and the simulator, and can be classified into *state*, *input*, and *output* variables [14]. State variables convey the system's configuration at a given point in time, and reflect the level of detail chosen by the modeler while developing the conceptual model. For example, a basic model of a queueing system may have only one state variable counting the elements waiting to be processed. Instead, a more detailed model may also have additional variables tracking how long each element has stayed in the queue. Input variables represent external influences on the system and serve as a source of information coming from the environment surrounding the system itself. For instance, the room temperature in which an electric heater operates could be an input variable in its model. Output variables provide insights into the system's behavior. Their definition strongly depends on the purpose of the simulation, on why the conceptual model has been developed in the first place. They might represent some physical dimension, such as power consumption or flow rate, but they also might implement more abstract concepts, like statistical measurements or decision outcomes. Following the previous examples, power consumption may be an output variable for the electric heater model, while the average waiting time may be an output variable for the queueing system.

The tools and languages selected for developing a simulator play a significant role in shaping the process. General-purpose *programming languages* offer a high degree of flexibility, allowing the modeler to implement the simulation's behavior directly

through algorithms and data structures. This flexibility supports hardware-specific optimizations and enables efficient simulations. However, achieving both correctness and efficiency can be challenging, as it demands a wide-ranging expertise that encompasses mathematical principles as well as hardware architectures.

Simulation languages provide a higher level of abstraction by limiting the range of available features and customizations in favor of simulation-specific constructs [65]. For example, the GPSS language [28] provides the `GENERATE` statement to represent the occurrence of events having a certain periodicity and standard deviation in time. Its syntax – taking only two numbers describing such parameters – can be clearly understood by the modeler and enables the rapid implementation of the conceptual model. At the same time, these constructs are typically general enough to cover multiple domains, thus providing a good balance between applicability and difficulty of adoption.

However, both programming and simulation languages require the modeler to manually transition from the descriptive nature of the conceptual model to the algorithmic structure required by the simulator [3]. This challenge has been addressed with the introduction of *modeling languages*, which provide syntax and semantics that are closely aligned with the conceptual models. The key innovation of modeling languages consists in the automation of transforming a declarative model into an executable simulator, a task that is performed by separately developed software solutions named *compilers*.

2.3 Foundations of Continuous Dynamic Systems

In this manuscript, we focus on models that are continuous and deterministic. Discrete and combined systems, even though supported by the modeling language of interest – introduced later in Section 2.5 – are not part of the initial development of the compiler being presented, and their theoretical aspects are therefore not discussed. Stochastic systems are also not considered in this initial stage, as not supported by the language, and thus would require syntactic and semantic extensions going out of the scope of this work. Therefore, in this section, we analyze the mathematical concepts that are essential to the process of modeling continuous dynamic systems — from now on simply referred to as *continuous systems*. Continuous systems are governed by differential equations [14, 45], of which we now provide a brief overview.

An Ordinary Differential Equation (ODE) can be expressed in the generic form shown in (2.1), where x is the independent variable (usually *time*), y is the dependent variable, and F is a function involving x, y, and the derivatives of y. The number n represents the order of the ODE.

$$F(x, y(x), \frac{dy(x)}{dx}, \ldots, \frac{d^n y(x)}{dx^n}) = 0 \tag{2.1}$$

Similarly, a number of m ODEs form a system of equations in the form shown in (2.2), where $\mathbf{y}$ is a vector of m dependent variables $(y_1(x) \ldots y_m(x))$, $\mathbf{F}$ is a vector of m functions $(f_1(x, y, y') \ldots f_m(x, y, y'))$, and $\mathbf{0}$ is a zero-valued vector again having m elements.

$$\mathbf{F}(x, \mathbf{y}(x), \frac{d\mathbf{y}(x)}{dx}, \ldots, \frac{d^n \mathbf{y}(x)}{dx^n}) = \mathbf{0} \tag{2.2}$$

An ODE of arbitrary order n can be rewritten into a system of ODEs all having an order equal to one. As exemplified in (2.3), this modification can be trivially performed

by introducing $n-1$ additional differential variables and ODEs, and by replacing the higher-order derivatives with usages of such new variables.

$$y'' + 2y' + y = 0 \Rightarrow \begin{cases} v = y' \\ v' + 2v + y = 0 \end{cases} \tag{2.3}$$

For this reason, ODE systems are typically assumed to be composed of differential equations having an order at most equal to one. Their generic form is shown in (2.4), with x being again the independent variable, and $\mathbf{y}$ the vector of differential variables.

$$\mathbf{F}(x, \mathbf{y}(x), \mathbf{y}'(x)) = \mathbf{0} \tag{2.4}$$

The simulation of ODE systems, as well as in general of differential equations, can be performed by either computing an analytical solution or by applying numerical methods. The former approach leads to exact solutions but is applicable only to a limited number of well-studied cases, while the latter can be applied to a broader range of equations at the cost of approximations in the computed behavior.

Numerical integration algorithms work by iteratively advancing the simulated time by a certain quantity – either varying or fixed – and computing the updated state of the system based on the current value of state variables and of their derivatives. The Runge-Kutta family of methods, to which the famous explicit and implicit Euler methods belong, is an example of such an algorithm.

DAE systems represent a more general case of ODE systems in which some equations are *algebraic*, that is, they do not contain any derivative. This kind of equations is introduced to impose additional constraints on the differential variables, and represents a common outcome of physical systems modeling in engineering [32, 33].

The simulation of DAE systems is more sophisticated with respect to the ODE case and requires to first introduce the concept of *index*. The definition of *index* is unfortunately not trivial, as it is used with different meanings across the literature [15, 16]. Here, we follow the definition used by Cellier and Koffman [20], according to which the *index* of a system acts as an indicator of the constraints that exist among the equations. An ODE system is an index-0 DAE system, an index-1 DAE system contains algebraic loops but no structural singularities (i.e., the causalization process, which will be explained later, can be performed successfully), while higher-index systems contain structural singularities.

DAE systems with an index higher than 1 typically need to be rewritten in an index-1 form to be solved. This process – called *index reduction* – can be performed in an automated way, for example by applying the Pantelides algorithm [55, 19]. Techniques for numerically integrating high-index systems do exist, but are limited to systems of equations having very specific forms (e.g., Hessenberg form [37, 12]). Instead, due to the recurrence of DAE systems in the modeling environment, general solvers have been developed for index-1 DAE models. The IDA software library, part of the SUNDIALS suite [35], represents an example of such solvers. Alternatively, the simulation of an index-1 DAE system can be performed by reducing it to a system of ODEs through the process of *causalization*, and then applying an ODE solver. Causalization consists of a multi-step symbolic manipulation which involves at least the *matching* and *scheduling* steps: the former associates each equation to a variable, and the latter attempts to find an order between the equations – or the strongly connected components, if any –

while respecting the dependencies between variables. Choosing between DAE solvers and causalization is a model-dependent decision. DAE solvers can reduce the number of Newton iterations needed to solve implicit equations and algebraic loops, but may nonetheless lead to numerical errors in case of small time steps [20].

Another key difference between ODEs and DAEs lies in the initialization of the system, that is the computation of the $\mathbf{y}(x_0)$ and $\mathbf{y}'(x_0)$ set of values – with x_0 being the value of the independent variable at the beginning of the simulation – from which the numerical integration algorithm can then start operating. In the case of ODE systems, initial conditions are provided by the modeler for all the variables, and such a complete list of constraints allows the model translator to trivially generate the list of assignments leading to a consistent initial state. In the case of DAE systems, the acceptable values are, however, restricted by hidden constraints imposed by the algebraic equations, whose derivatives limit the set of values that the variables can evaluate to. Unveiling and handling such hidden constraints often represents a difficult task within both the translation pipeline – due to the required code manipulation capabilities – and the actual simulation – due to the possible non-convergence of the chosen numerical algorithm [74].

These key differences make the handling of DAEs worth of detailed consideration, and the literature on the topic is nonetheless generous in quantity [57, 20, 13].

Despite their difference in the resolution and simulation perspectives, both ODEs and DAEs are characterized by a unique independent variable – named x in (2.4) – with respect to which differentiation is performed. A generalization of ODEs is provided by Partial Differential Equations (PDEs), which allow for multiple independent variables, and partial derivatives to be performed with respect to each of them. Similarly to ODEs, analytical solutions can sometimes be obtained (e.g., through the method of separation of variables). However, due to their broader applicability, numerical methods are again often preferred.

The fundamental approach behind numerical methods for PDEs is to transform them into a set of coupled ODEs, which operate on the discretized space of the all-but-one independent variables. Typically, the discretized space corresponds to the three spatial dimensions, and the remaining independent variable represents time.

Well-known techniques for solving PDEs include the Finite Difference Method (FDM), Finite Element Method (FEM), and Finite Volume Method (FVM) [24]. Without delving into excessive detail, FDM involves dividing the domain into a structured grid – typically of rectangular shape, though it can be projected on more complex geometries – and approximating the PDEs locally at each nodal point using a Taylor series expansion. In contrast, FEM and FVM partition the spatial domain into simple geometries (e.g., triangles) and introduce additional algebraic equations binding these partitions together. In the case of FVM, these additional constraints usually consist of flux equations.

The choice of the discretization method is a model-dependent decision that is typically left to the modeler, whose knowledge of the application domain and its related mathematical literature can guide toward the most suitable solution. Of course, other discretization algorithms (e.g., orthogonal collocation of finite elements) are also available. However, for the sake of brevity, further discussion is omitted. The readers interested in a more in-depth analysis of numerical approximation of PDEs may refer to the

work of Quaternoni and Valli [59].

2.4 Software Solutions for Continuous Systems

This section provides an overview of the software solutions and modeling abstractions used to simulate continuous systems. The term *software solution* is used in a broad sense and includes both languages and simulation tools. Indeed, the two concepts are sometimes intertwined, as modeling and simulation languages may be accompanied by a one and only simulation tool that implements the semantics of the language itself, thus blurring the line that separates the two domains.

Table 2.1 illustrates the degree of compatibility of various tools and libraries with respect to the different types of differential equations discussed earlier. The focus is on solutions that are actively maintained or widely adopted in the industry. As anticipated in Section 2.1, the focus is on multi-domain modeling and simulation of physical and cyber-physical systems, with particular attention to Julia, MATLAB, and Modelica.

From a modeling perspective, two primary approaches can be identified. One requires the modeler to explicitly define the vectors and matrices of coefficients describing the system's behavior, and call specific functions – with well-defined parameters – to apply the desired numerical integration methods. The second approach, instead, provides higher levels of abstraction and reduces the amount of mathematical manipulations that the modeler needs to perform beforehand.

MATLAB exemplifies the first approach by offering dedicated functions solving ODEs, DAEs, and a limited set of PDEs. Julia's DifferentialEquations library [61] also keeps the focus on the numerical side of the problem, and leaves to complementary libraries the task of implementing the higher-level abstractions that are typical of the second approach. The ModelingToolkit [49] and Modia [23] libraries are examples of such an extension. Other languages and tools, however, directly adopt the second approach. In Modelica and gPROMS, for example, the time derivative operator is built directly into the language.

Regarding the application of numerical integration algorithms and the corresponding generation of the simulator program, two opposite approaches can be again identified within the individual tools and libraries supporting ODEs and DAEs.

On the one hand, simple numerical integration algorithms are often implemented internally. This is, for example, the case of the Runge-Kutta family of algorithms, which includes the famous explicit and implicit Euler methods. On the other hand, more complicated algorithms are typically made available through external software libraries, such as the CVODE solver – used, among others, by OpenModelica – and Lsoda – used by Altair Activate – for simulating ODE systems. The integration with external solvers is also a well-established practice in the case of DAE systems, for which the complexity of the involved algorithms is higher. The SUNDIALS IDA and DASSL [56] solvers – both targeting DAE systems of index 1 – are used by the majority of the existing solutions, and their long-lived existence represents a reasonable guarantee of correctness.

In the case of PDEs, many software solutions have been independently developed throughout the last decades, in both open-source (e.g., OpenFOAM [36]) and proprietary forms. However, these tools are typically tailored to specific application domains

Table 2.1: *Characteristics of common tools and libraries operating in the field of continuous systems. For each software solution, the table reports its kind, the interface available to the modeler, the underlying technology, and the level of support of tools and libraries for different kinds of equation systems. The checkmark (✓), approximate (≈), and cross (×) symbols respectively indicate a mature level of adoption, an approximate or experimental support, and the unavailability of the feature.*

Software	Kind	Interface	Technology	ODE	DAE	PDE
Altair Activate	Tool	Graphical	Proprietary	✓	✓	×
ANSYS TwinBuilder	Tool	Textual (Modelica)	Proprietary	✓	✓	×
COMSOL Multiphysics	Tool	Graphical	Proprietary	✓	✓	✓
CoSApp	Library	Textual (Python)	Python	✓	✓	×
DifferentialEquations.jl	Library	Textual (Julia)	Julia	✓	✓	✓
Dymola	Tool	Textual (Modelica) Graphical	Proprietary	✓	✓	×
EcosimPro	Tool	Graphical	Proprietary	✓	✓	×
gPROMS	Tool	Textual (DSL) Graphical	Proprietary	✓	✓	✓
MapleSim	Tool	Textual (Modelica) Graphical	Maple	✓	✓	×
ModelingToolkit.jl	Library	Textual (DSL)	Julia	✓	✓	✓
Modelon Impact	Tool	Textual (Modelica, Optimica) Graphical	OPTIMICA Compiler Toolkit	✓	✓	×
Modia.jl	Library	Textual (DSL)	Julia	✓	✓	×
OpenModelica	Tool	Textual (Modelica) Graphical	MetaModelica	✓	✓	≈
Scicos	Tool	Textual (C, Fortran, Modelica, Scilab) Graphical	Scilab	✓	✓	×
Simcenter Amesim	Tool	Textual (C, Modelica, Python) Graphical	Proprietary	✓	✓	×
Simscape	Tool	Textual (DSL) Graphical	MATLAB	✓	✓	×
SimulationX	Tool	Textual (Modelica) Graphical	Proprietary	✓	✓	×
Simulink	Tool	Graphical	MATLAB	✓	✓	×
Wolfram System Modeler	Tool	Textual (Modelica) Graphical	Proprietary	✓	✓	×

(e.g., CFD [39]) that are outside the scope of cyber-physical systems modeling.

In this domain, system-level representation is generally not conceptualized in terms of PDEs, but rather as the interaction between the components of the system. While some components may be modeled using PDEs (e.g., heat exchangers), their implementation often relies on a discretization performed manually by the modeler. Given that these components frequently exhibit regular (e.g., rectangular) geometric shapes, the need for sophisticated meshing algorithms is generally minimal. Nevertheless, certain languages and software tools do provide built-in support for modeling and simulating PDEs.

2.5 Modelica and Base Modelica

Modelica is a non-proprietary, equation-based, language for modeling complex physical systems. It is designed to support the modeling of both continuous and discrete dynamics, making it suitable for a wide range of applications in engineering and scientific computing. It is characterized by declarative, acausal, semantics, which allow for the creation of digital representations that closely mirror the conceptual model of the system being studied.

In order to ease the understanding of the language features, let us consider the simple electrical circuit of Figure 2.1, for which the Modelica representation is shown in Listing 2.1.

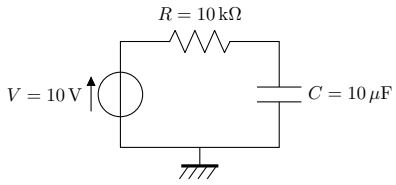

Figure 2.1: *Schematic of an example RC circuit with a constant voltage source, a resistor, and a capacitor connected in series.*

The key construct in the Modelica syntax is the *model*. For the purpose of this manuscript, models are assumed to be composed of only two blocks, the first for declaring variables, parameters, and constants, and the second for listing the equations. Moreover, the examples and explanations in this work consider only continuous semantics, as discrete semantics go beyond the scope of the currently implemented solution.

The acausal nature of the language can be observed in the declaration of equations. Indeed, the `=` operator represents an equality in the mathematical sense, and not an assignment. For example, Ohm's law may be written as `v = r * i` as well as `0 = v - r * i`, without implying any semantical or behavioral change. Similarly, the order in which equations are listed within a model also does not influence the model's behavior.

Modelica is also object-oriented and component-based. The former means that models can be defined in term of classes, which can then be extended and customized. The latter allows for composing models from smaller, reusable components, which can be instantiated and connected together to form bigger systems. In the example of Listing 2.1, the voltage source, ground, resistor, and capacitor objects are described as independent classes which are then instantiated in the circuit model. The `connect` statements describe how the components are interconnected. For example, the first `connect`

```
package Electrical
  connector Pin
    Real v;
    flow Real i;
  end Pin;

  connector TwoPin
    Pin p;
    Pin n;
    Real v;
  equation
    v = p.v - n.v;
  end TwoPin;

  connector Port extends TwoPin;
    Real i;
  equation
    0 = p.i + n.i;
    i = p.i;
  end Port;

  model VoltageSource extends Port;
    parameter Real V;
  equation
    v = V;
  end VoltageSource;

  model Ground
    Pin p;
  equation
    p.v = 0;
  end Ground;

  model Resistor extends Port;
    parameter Real R;
  equation
    v = R * i;
  end Resistor;

  model Capacitor extends Port;
    parameter Real C;
  equation
    i = C * der(v);
  end Capacitor;

  model Circuit
    VoltageSource source(V = 10);
    Ground gnd;
    Resistor r(R = 10e3);
    Capacitor c(C = 10e-6, v(start = 0, fixed = true));
  equation
    connect(source.p, r.p);
    connect(r.n, c.p);
    connect(c.n, source.n);
    connect(source.n, gnd.p);
  end Circuit;
end Electrical;
```

Listing 2.1: *Modelica representation of the RC circuit of Figure 2.1.*

statement connects the positive terminal of the voltage source to one of the two terminals of the resistor. The second `connect` statement then connects the other terminal of the resistor to one of the two terminals of the capacitor, and so on.

Combined with the acausal nature of the language, object orientation, and component-based design allow for a high level of reusability and to produce a codified representation that closely mirrors the conceptual model. Considering again the example of Listing 2.1, the model of a resistor can be defined independently of how it will be used in a circuit. There is no need to specify whether the input is the voltage at its terminal contacts or the current flowing through it. In other words, it is the job of the compiler, and not of the user, to symbolically manipulate Ohm's law according to the context in which the resistor is used.

However, while the Modelica language is meant to be written by humans and therefore includes constructs such as object orientation and connect statements to describe models in an expressive way, the Modelica community recently acknowledged the need for a simplified version of the language, tailored for automated machine code generation.

This language, named Base Modelica [41], is currently in the process of being standardized. Tools already exist to convert Modelica code to Base Modelica, thus making it a viable source language to research advanced compilation techniques languages based on DAEs. Indeed, Base Modelica allows focusing on the compiler algorithms and data structures without the need to design a frontend capable of handling the complexity of the entire Modelica language specification [53].

The circuit of Listing 2.1, when flattened into Base Modelica code, reads as Listing 2.2. It should be noted how the class hierarchy and component instantiations have been expanded into variables describing each quantity, all in a single model, and the interconnection constructs have been replaced by the equations they stood for.

2.6 Limitations of State of the Art Modelica Compilers

In Section 2.5, we examined a simple RC circuit and its Base Modelica representation of Listing 2.2. This example consisted of a scalar model, meaning that all variables and equations are defined for single quantities and do not involve array constructs. However, many physical systems are naturally described by arrays of variables, especially when discretizing spatial domains. Let us now consider the thermal model of a solid copper wire divided into $N = 1000$ sections of equal length. Each partition is represented by its temperature, and heat conduction between adjacent partitions is modeled through a set of equations. In Base Modelica, such a model can be conveniently expressed using the `for` construct, as shown in Listing 2.3.

The usage of the `for` construct allows the compact expression of multiple equations that share a common structure – in this case, the heat conduction equations and the initial conditions for each partition – without the need to manually write each scalar equation. By modifying the size of the array variable `T` and the boundaries of the `for` loops, the granularity of the model can be easily adjusted without leading to a modification of the memory footprint of its digital representation.

The proposed one-dimensional thermal model has been used by Agosta et al. [5] to evaluate both the compilation and simulation performance of industry-grade com-

```
package 'Circuit'
  model 'Circuit'
    Real 'source.p.v';
    Real 'source.p.i';
    Real 'source.n.v';
    Real 'source.n.i';
    Real 'source.v';
    Real 'source.i';
    parameter Real 'source.V' = 10.0;
    Real 'gnd.p.v';
    Real 'gnd.p.i';
    Real 'r.p.v';
    Real 'r.p.i';
    Real 'r.n.v';
    Real 'r.n.i';
    Real 'r.v';
    Real 'r.i';
    parameter Real 'r.R' = 1e4;
    Real 'c.p.v';
    Real 'c.p.i';
    Real 'c.n.v';
    Real 'c.n.i';
    Real 'c.v'(fixed = true, start = 0.0);
    Real 'c.i';
    parameter Real 'c.C' = 1e-5;
  equation
    'source.p.v' = 'r.p.v';
    'r.n.v' = 'c.p.v';
    'source.n.v' = 'gnd.p.v';
    'source.n.v' = 'c.n.v';
    'r.p.i' + 'source.p.i' = 0.0;
    'c.p.i' + 'r.n.i' = 0.0;
    'c.n.i' + 'gnd.p.i' + 'source.n.i' = 0.0;
    'source.v' = 'source.V';
    0.0 = 'source.p.i' + 'source.n.i';
    'source.i' = 'source.p.i';
    'source.v' = 'source.p.v' - 'source.n.v';
    'gnd.p.v' = 0.0;
    'r.v' = 'r.R' * 'r.i';
    0.0 = 'r.p.i' + 'r.n.i';
    'r.i' = 'r.p.i';
    'r.v' = 'r.p.v' - 'r.n.v';
    'c.i' = 'c.C' * der('c.v');
    0.0 = 'c.p.i' + 'c.n.i';
    'c.i' = 'c.p.i';
    'c.v' = 'c.p.v' - 'c.n.v';
  end 'Circuit';
end 'Circuit';
```

Listing 2.2: *Base Modelica representation of the RC circuit of Figure 2.1, as obtained after the instantiation of the Circuit model of Listing 2.1. Object-orientation and component connections are lost, and a unique model is obtained.*

```
1  package 'Wire'
2    model 'Wire'
3      parameter Real 'g' = 0.00314785;
4      parameter Real 'c' = 0.2707936;
5      Real 'T'[1000];
6    equation
7      for 'i' in 2 : 999 loop
8        'c' * der('T'['i']) = 'g' * ('T'['i' - 1] - 'T'['i']) +
9                              'g' * ('T'['i' + 1] - 'T'['i']);
10     end for;
11
12     'c' * der('T'[1]) = 'g' * ('T'[2] - 'T'[1]) +
13                         2 * 'g' * (673.15 - 'T'[1]);
14
15     'c' * der('T'[1000]) = 'g' * ('T'[999] - 'T'[1000]) +
16                            2 * 'g' * (293.15 - 'T'[1000]);
17   initial equation
18     for 'i' in 1 : 1000 loop
19       'T'['i'] = 293.15;
20     end for;
21   end 'Wire';
22 end 'Wire';
```

Listing 2.3: *Base Modelica representation of a thermal model of a copper wire divided into 1000 partitions, using an array variable and a* `for` *equation to represent the partitions and the heat conduction acting between adjacent ones.*

pilers – OpenModelica and Dymola – while changing the parameterized number of partitions, and therefore increasing the number of scalar variables and equations of the system. The comparison with a hand-written C++ simulator – in which arrays are considered as first-class citizens – demonstrated the inability of such tools to properly scale with respect to the parametric size of the model, and a shift of paradigm was deemed necessary to enable the simulation of large-scale models.

One of the most critical issues is caused by the so-called *flattening* of array variables and array equations. During the compilation process, arrays are expanded into individual scalar variables and equations. Considering the previous example of Listing 2.3, its flattened representation would result in Listing 2.4 – which is actually trimmed for space reasons – where each array variable is replaced by a series of scalar variables, and the `for` equation is expanded into multiple scalar equations. Notice how the variable names would also include the index of the array, which would be, by all means, treated as part of the identifier, thus losing the array structure. While this approach simplifies the implementation of certain analyses (e.g., causalization), it also results in excessive memory usage and unbearable compilation times [6].

Moreover, the architecture of current compilers is rooted in traditional compiler design, which is increasingly inadequate for the demands of modern modeling languages. OpenModelica is a bootstrapping compiler, meaning it is implemented in a language that compiles itself. To achieve this capability, the Modelica language was extended to MetaModelica, introducing features such as lists and exceptions that are not part of standard Modelica. While this approach enabled rapid development and prototyping, it is hard to justify such a choice considering the purpose of Modelica itself, which is not

```
package 'Wire'
  model 'Wire'
    parameter Real 'g' = 0.00314785;
    parameter Real 'c' = 0.2707936;
    Real 'T[1]';
    Real 'T[2]';
    ...
    Real 'T[1000]';
  equation
    'c' * der('T[2]') = 'g' * ('T[1]' - 'T[2]') +
                        'g' * ('T[3]' - 'T[2]');

    'c' * der('T[3]') = 'g' * ('T[2]' - 'T[3]') +
                        'g' * ('T[4]' - 'T[3]');

    ...

    'c' * der('T[999]') = 'g' * ('T[998]' - 'T[999]') +
                         'g' * ('T[1000]' - 'T[999]');

    'c' * der('T[1]') = 'g' * ('T[2]' - 'T[1]') +
                        2 * 'g' * (673.15 - 'T[1]');

    'c' * der('T[1000]') = 'g' * ('T[999]' - 'T[1000]') +
                           2 * 'g' * (293.15 - 'T[1000]');
  initial equation
    'T[1]' = 293.15;
    'T[2]' = 293.15;
    ...
    'T[1000]' = 293.15;
  end 'Wire';
end 'Wire';
```

Listing 2.4: *Flattened version of the Base Modelica model of Listing 2.3, where the array variable* `T` *and the* `for` *equations have been expanded into individual scalar variables and equations.*

designed as a general-purpose programming language like C or C++. The remaining Modelica compilers are closed-source (e.g., Dymola, Wolfram SystemModeler), making it difficult to draw conclusions about their internal architectures, and at the same time limiting opportunities for collaboration and innovation within the community.

In response to the limitations of existing Modelica compilers, this thesis proposes the advances achieved within the Modelica Advanced Research COmpiler (MARCO) to enable the efficient compilation and simulation of DAE systems. This objective is pursued with respect to three aspects, which are covered in the following chapters. In the first place, the attention is focused on the implementation of algorithms and data structures that serve as the mathematical backbone for the efficiency of the whole compiler. The MLIR framework is then used as a foundation of both the compiler and its Intermediate Representation (IR) to obtain a modular, extensible, and interoperable solution for compilation, able to preserve high-level constructs that, in cooperation with the above algorithms, enable constant-time compilation with respect to the parametric size of arrays. The design of the compiler IRs is also performed in conjunction with the implementation of an array-aware runtime environment, whose structure and algorithms are finally detailed.

CHAPTER 3

Array-Aware Compilation

In Modelica, array variables and equations are defined over a multidimensional space of points, where each point corresponds to a particular combination of indices. Efficiently representing and operating on these multidimensional index spaces is crucial for both the analysis and the transformation of the model, especially in the large-scale context that has been motivating the development of MARCO since its inception. For this reason, this chapter is dedicated to the improvements to the state-of-the-art algorithms and data structures used for the array-aware compilation of DAE systems.

3.1 Multidimensional Space Indexing

In this section, we address the problem of efficiently representing and manipulating multidimensional index spaces. We describe the involved data structures and algorithms, starting from the basic building blocks of points and ranges, and then moving to the more complex implementation of an R-Tree-based set of indices.

Point

The *Point* class represents a single point in a multidimensional space, with one integer coordinate per dimension. It is used to denote specific indices in array variables and equations. On a mathematical level, we denote a point as a tuple $p = (p_1, p_2, \ldots, p_n)$, where $n \in (\mathbb{N} \setminus \{0\})$ is the *rank* – that is the number of dimensions – of point p, and $p_i \in \mathbb{Z}$ is the coordinate along the i-th dimension.

Range

The *Range* class encapsulates a contiguous interval of integer values along a single dimension, defined by a lower bound $l \in \mathbb{Z}$ and an upper bound $u \in \mathbb{Z}$ such that $l \leq u$.

Each index i contained by the range satisfies the condition $l \leq i < u$. Empty ranges are those where $l = u$. For convenience, a range r is also shortly represented as $[l, u)$. We also define r_L and r_U as the shorthands for the lower and upper bounds of a range r.

This representation provides efficient storage and manipulation capabilities to handle the array variables and array equations that frequently appear in Modelica models. Range comparisons are performed based on the boundary values, enabling fundamental operations such as containment verification and intersection computation to be executed in constant time complexity with respect to the actual number of points.

The Range class supports a comprehensive set of operations, including point membership testing, overlap detection, and range arithmetic. These operations form the foundational primitives for the more sophisticated Multidimensional Range and Index Set data structures described subsequently. The complete specification of range operations is presented in Algorithm 1 and Algorithm 2.

Algorithm 1 Range Operations (1/2)

```
// Check if a range is empty (i.e., contains no points).
function EMPTY(r: Range)
    return r_L = r_U
end function

// Check if ranges r_1 and r_2 have any common points.
function OVERLAP(r_1: Range, r_2: Range)
    return ¬ EMPTY(r_1) ∧ ¬ EMPTY(r_2) ∧ r_1L ≤ (r_2U − 1) ∧ (r_1U − 1) ≥ r_2L
end function

// Check if range r_2 is entirely contained within range r_1.
function CONTAIN(r_1: Range, r_2: Range)
    return EMPTY(r_2) ∨ (r_1L ≤ r_2L ∧ r_1U ≥ r_2U)
end function

// Compute the intersection of two ranges r_1 and r_2.
function INTERSECT(r_1: Range, r_2: Range)
    if ¬ OVERLAP(r_1, r_2) then
        return [0, 0)
    end if
    if CONTAIN(r_1, r_2) then
        return r_2
    else if CONTAIN(r_2, r_1) then
        return r_1
    else if r_1L ≤ r_2L then
        return [r_2L, r_1U)
    else
        return [r_1L, r_2U)
    end if
end function
```

Multidimensional Range

The *MultidimensionalRange* class represents a hyper-rectangular region in multidimensional space, defined as $r = r_1 \times \ldots \times r_n$, where $n \in (\mathbb{N} \setminus \{0\})$ and each r_i is a *Range*

Algorithm 2 Range Operations (2/2)

```
// Get the ranges whose points are in the range r_1 but not in the range r_2.
function SUBTRACT(r_1: Range, r_2: Range)
    result ← ∅
    if ¬ OVERLAP(r_1, r_2) then
        result ← result ∪ r_1
    else if CONTAIN(r_1, r_2) then
        if r_{1L} ≠ r_{2L} then
            result ← result ∪ [r_{1L}, r_{2L})
        end if
        if r_{1U} ≠ r_{2U} then
            result ← result ∪ [r_{2U}, r_{1U})
        end if
    else if ¬ CONTAIN(r_2, r_1) then
        if r_{1L} ≤ r_{2L} then
            result ← result ∪ [r_{1L}, r_{2L})
        else
            result ← result ∪ [r_{2U}, r_{1U})
        end if
    end if
    return result
end function

// Merge two ranges r_1 and r_2 into a single range.
// If r_1 and r_2 cannot be merged (i.e., they are disjoint, non-adjacent, and neither is empty), the function returns an empty range.
function MERGE(r_1: Range, r_2: Range)
    if EMPTY(r_1) then
        return r_2
    else if EMPTY(r_2) then
        return r_1
    else if OVERLAP(r_1, r_2) ∨ r_{1L} = r_{2U} ∨ r_{1U} = r_{2L} then
        return [MIN(r_{1L}, r_{2L}), MAX(r_{1U}, r_{2U}))
    else
        return [0, 0)
    end if
end function
```

object representing the bounds of the range in the i-th dimension.

Like the *Point* class, the dimensionality of the space is termed the *rank*, and operations between multidimensional ranges – or between multidimensional ranges and points – are only valid for entities sharing the same rank.

The *MultidimensionalRange* preserves the contiguous property of its constituent *Range* objects, and therefore all points within the Cartesian product of the individual ranges are included in the represented set. For example, a rank-2 multidimensional range composed of ranges $[1, 3)$ and $[2, 5)$ represents all and only the points of the set $\{(1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4)\}$, thus encompassing all combinations of coordinates within the specified bounds.

The fundamental operations supported by multidimensional ranges semantically mirror those of the single-dimensional *Range* class, as detailed in Algorithm 3 and Algorithm 4. Comparisons between multidimensional ranges are performed dimension-wise, evaluating the lower and upper bounds of each constituent range. Therefore, the computational complexity of these operations scales linearly with the rank.

Algorithm 3 Multidimensional Range Operations (1/2)

```
1:  // Check if multidimensional ranges r_1 and r_2 have any common points.
2:  function OVERLAP(r_1: MultidimensionalRange, r_2: MultidimensionalRange)
3:      for i ← 0 to RANK(r_1) −1 do
4:          if ¬ OVERLAP(r_1[i], r_2[i]) then
5:              return false
6:          end if
7:      end for
8:      return true
9:  end function
10:
11: // Check if multidimensional range r_2 is entirely contained within multidimensional range r_1.
12: function CONTAIN(r_1: MultidimensionalRange, r_2: MultidimensionalRange)
13:     for i ← 0 to RANK(r_1) −1 do
14:         if ¬ CONTAIN(r_1[i], r_2[i]) then
15:             return false
16:         end if
17:     end for
18:     return true
19: end function
20:
21: // Compute the intersection of two overlapping multidimensional ranges.
22: function INTERSECT(r_1: MultidimensionalRange, r_2: MultidimensionalRange)
23:     ranges ← [ ]
24:     for i ← 0 to RANK(r_1) −1 do
25:         ranges.append(INTERSECT(r_1[i], r_2[i]))
26:     end for
27:     return MULTIDIMENSIONALRANGE(ranges)
28: end function
```

R-Tree & Index Set

Up to this point, we have described the basic data structures that are used to represent points and ranges in a multidimensional space. However, we still have to address

Algorithm 4 Multidimensional Range Operations (2/2)

```
29: // Get the ranges whose points are in r_1 but not in r_2.
30: function SUBTRACT(r_1: MultidimensionalRange, r_2: MultidimensionalRange)
31:     result ← ∅
32:     if ¬ OVERLAP(r_1, r_2) then
33:         result ← result ∪ r_1
34:     else
35:         currentRanges ← r_1.ranges
36:         for i ← 0 to RANK(r_1) −1 do
37:             for subRange in SUBTRACT(r_1[i], r_2[i]) do
38:                 currentRanges[i] ← subRange
39:                 result ← result ∪ MULTIDIMENSIONALRANGE(currentRanges)
40:             end for
41:             currentRanges[i] ← INTERSECT(r_1[i], r_2[i])
42:         end for
43:     end if
44:     return result
45: end function
46:
47: // Merge two multidimensional ranges r_1 and r_2 into a single range.
48: // If they cannot be merged (i.e., they are disjoint or adjacent on more than one dimension), the
    function returns an empty range.
49: function MERGE(r_1: MultidimensionalRange, r_2: MultidimensionalRange)
50:     for i ← 0 to RANK(r_1) −1 do
51:         if EMPTY(r_1[i]) then
52:             return r_2
53:         else if EMPTY(r_2[i]) then
54:             return r_1
55:         end if
56:     end for
57:     mergedRanges ← [ ]
58:     numMerged ← 0
59:     for i ← 0 to RANK(r_1) −1 do
60:         if r_1[i] = r_2[i] then
61:             mergedRanges.append(r_1[i])
62:         else
63:             mergedRange ← MERGE(r_1[i], r_2[i])
64:             if ¬ EMPTY(mergedRange) ∧ numMerged = 0 then
65:                 mergedRanges.append(mergedRange)
66:                 numMerged ← numMerged +1
67:             else
68:                 mergedRanges.append([0, 0))
69:             end if
70:         end if
71:     end for
72:     return MULTIDIMENSIONALRANGE(mergedRanges)
73: end function
```

the problem of efficiently indexing and querying these multidimensional entities when dealing with large sets of spatially-disjoint data. A naive approach would be to use a simple list to store points and ranges, but this would lead to inefficient search operations, as it would require scanning the entire list to find a specific element. In the context of MARCO, this problem is addressed by relying on a more sophisticated data structure known as the *R-Tree*.

As the name suggests, an R-Tree consists in a tree-like structure where each node is characterized by a hyper-rectangular bounding box that encloses all of its children. This hierarchical structure allows for efficient spatial querying, as it enables the R-Tree to typically ignore large portions of the search space that do not intersect with the query region.

In MARCO, two implementations of the R-Tree are used, one for indexing generic objects, and one specialized for indexing multidimensional points and ranges. The former is built on the original algorithms proposed by Guttman [31] and aims to efficiently index any kind of object for which a multidimensional boundary can be defined. Its implementation – not bringing considerable novelty – is not detailed in this work. The latter, named *Index Set*, is specifically optimized for indexing multidimensional points and ranges, and, as a result, brings additional features and optimizations that arise from the known leaf data-type.

The *Index Set* behaves as a set of multidimensional points, which are however stored internally in a compact way through the usage of multidimensional ranges as R-Tree leaves. Differently from the general R-Tree data structure, which has an opaque vision of the leaves, the Index Set reasons about the individual points belonging to the ranges.

The iteration over the points – or ranges – as well as their counting, thus requires to keep into account possible duplicates among the leaf ranges. However, this difficulty is addressed directly during the insertion of new elements, whose algorithm is extended to first remove, from the inserted ranges, the points that already belong to the set. The algorithm for range insertion is detailed in Algorithm 5.

In Algorithm 5, the `getRanges` function returns the list of ranges stored in the R-Tree, while the `insertObject` function inserts a new range into the underlying R-Tree data structure, thus treating the range as a whole. These two functions are not detailed here: the former consists of a simple post-order traversal of the R-Tree, while the latter is already described in the original R-Tree implementation [31].

The removal of a range of points is detailed in Algorithm 6. Similarly to insertion, this procedure is also influenced by the point semantics of the IndexSet data structure. When removing a multidimensional range, its points may be spread across multiple leaf nodes, and therefore the removal operation must traverse all the nodes whose bounding rectangle overlaps with the range to be removed. This traversal is implemented by the `getOverlappingLeafNodes` function. The `adjustTree` function is used to adjust the tree structure after a modification of a node's children, ensuring that the R-Tree properties are maintained. Both these last two functions are not detailed here, as they do not bring considerable novelty.

Algorithm 5 IndexSet: Range Insertion

```
1: // Insert the points contained in a multidimensional range r
2: function INSERT(r: MultidimensionalRange)
3:     nonOverlapping ← ∅
4:     if root = null then
5:         nonOverlapping ← nonOverlapping ∪ {r}
6:     else
7:         current ← {r}
8:         for r_i in GETRANGES() do
9:             next ← ∅
10:            for r_j in current do
11:                for r_k in SUBTRACT(r_j, r_i) do
12:                    if OVERLAP(r_k) then
13:                        next ← next ∪ {r_k}
14:                    else
15:                        nonOverlapping ← nonOverlapping ∪ {r_k}
16:                    end if
17:                end for
18:            end for
19:            current ← next
20:        end for
21:    end if
22:    for r_i in nonOverlapping do
23:        INSERTOBJECT(r_i)
24:    end for
25: end function
```

Algorithm 6 IndexSet: Range Removal

```
1: // Remove the points contained in a multidimensional range r
2: function REMOVE(r: MultidimensionalRange)
3:     overlapping ← GETOVERLAPPINGLEAFNODES(r)
4:     for node in overlapping do
5:         modified ← false
6:         differences ← ∅
7:         for r_i in node.children do
8:             if OVERLAP(r_i, r) then
9:                 modified ← true
10:                differences ← differences ∪ SUBTRACT(r_i, r)
11:            else
12:                differences ← differences ∪ {r_i}
13:            end if
14:        end for
15:        node.children ← differences
16:        if modified then
17:            ADJUSTTREE(node)
18:            break
19:        end if
20:    end for
21: end function
```

3.2 Mapping Equation to Variable Indices

The concepts of multidimensional space indexing described in Section 3.1 find an immediate application when considering the relationships between the equations and variables of an array-oriented DAE system.

In the context of this thesis, we informally define an *access function* as a mathematical mapping that describes the relationship between the indices of an equation and the indices of a variable referenced within that equation. If the variable is referenced multiple times, an access function is defined for each occurrence of the variable within the equation.

More formally, let $\mathcal{E}$ denote an array equation with indices $E \subset \mathbb{Z}^n$, where n is the rank – that is, the amount of coordinates – of $\mathcal{E}$, and let $\mathcal{V}$ denote an array variable with indices $V \subset \mathbb{N}^m$, where m is the rank of $\mathcal{V}$.

Given an occurrence of $\mathcal{V}$ in $\mathcal{E}$, the associated access function between $\mathcal{E}$ and $\mathcal{V}$ is defined as a relation $f : E \to V$ such that

$$(e, v) \in f \iff \mathcal{E}_e \text{ references } \mathcal{V}_v$$

with $\mathcal{E}_e$ and $\mathcal{V}_v$ respectively being the equation $\mathcal{E}$ and variable $\mathcal{V}$ at the indices e and v.

It is important to notice how, despite the chosen naming, f is not necessarily a function in the strict mathematical sense. An equation (multidimensional) index may relate to multiple scalar variables (e.g., in pseudo-code, `v[6 to 10]` would access a slice of 5 elements of variable `v`), and also the same variable (multidimensional) index may relate to multiple scalar equations (e.g., again in pseudo-code, `v[1]` would see the scalar variable at index 1 as accessed by all the equation indices).

For this reason, we provide an additional definition of *array access function*, which relates each set of equation indices $\hat{E} \subseteq E$ to a set of variable indices $\hat{V} \subseteq V$, in the form of a function $\bar{f} : 2^E \to 2^V$ – where 2^X is the power set of X (i.e., the set of all subsets of X) – such that

$$\bar{f}(\hat{E}) = \{v \mid v \in V \wedge \exists\, e \in \hat{E} : (e, v) \in f\}$$

The definition of $\bar{f}$ encompasses all possible combinations of equation indices, thus overcoming the limitation of the previous definition.

It is also useful to define the concept of *inverse array access function*, which informally allows, given a set of variable indices $\hat{V} \subseteq V$, to obtain the equation indices $\hat{E} \subseteq E$ that access those scalar variables. Formally, the inverse access function $\bar{g} = \bar{f}^{-1}$ is defined as a function $\bar{g} : 2^V \to 2^E$ such that

$$\bar{g}(\hat{V}) = \{e \mid e \in E \wedge \bar{f}(\{e\}) \subseteq \hat{V}\}$$

For clarity, we now establish some notation and terminology that will be used in the remaining part of this work. First, f and $\bar{f}$, independently of being a function or only a relation, will be referred to as *access functions*. When dealing with an access function $\bar{f}$ and its inverse $\bar{g}$, we will refer to $\hat{V} = \bar{f}(\hat{E})$ – with $\hat{E} \subseteq E$ – as being the *mapping* of $\hat{E}$ through $\bar{f}$, and we will refer to $\hat{E} = \bar{g}(\hat{V})$ – with $\hat{V} \subseteq V$ – as being the *inverse mapping* of $\hat{V}$ through $\bar{f}$. Depending on the context, the same wording will also be used to refer to the functions $\bar{f}$ and $\bar{g}$ themselves rather than to the sets of indices they compute.

Generic Implementation In MARCO, a basic implementation is provided to cover the most general kind of access function, allowing for the compilation of models with arbitrary accesses. Given a set of multidimensional indices of the equation, the mapping operation is implemented by applying $\bar{f}$ to each of the points, and by performing a set-union operation on each result. More formally, for an equation $\mathcal{E}$ with index space E and a reference to variable $\mathcal{V}$ of indices V, the mapping of indices $\hat{E} \subseteq E$ consists in the set

$$\hat{V} = \bigcup_{e \in \hat{E}} \bar{f}(\{e\})$$

Of course, the generic implementation of an access function is computationally inefficient, as it always necessitates iterating over all the points of the argument set. It is also important to note that no general formula can be devised to compute an inverse mapping, and an iteration over all points of the equation space is therefore required, as per the previous definition of $\bar{g}$.

Acknowledging the limitations of a generic access function implementation, ad-hoc versions are provided for specific cases that are deemed to often appear in DAE models, such as constant (e.g., `[1]`) and affine-like accesses (e.g., `[i + k]` and `[i - k]`, with k constant, which are common patterns resulting from spatial discretization of PDEs). These specialized implementations play a crucial role in speeding up the compilation process, as they allow for faster mapping and inverse mapping procedures.

Constant Access Function A *constant access function* is a special case where the mapping from equation indices to variable indices is fixed and does not depend on the specific equation index. Formally, a constant access function $\bar{f}_c$ is defined as

$$\bar{f}_c(\hat{E}) = V_k \quad \forall\, \hat{E} \subseteq E$$

with $V_k \subseteq V$ being a fixed set of variable indices given by the access itself (e.g., with access `[{1, 3, 5}]`, $V_k = \{(1), (3), (5)\}$). The computation of the inverse mapping is straightforward, as always equal to E:

$$\bar{g}_c(\hat{V}) = E \quad \forall\, \hat{V} \subseteq V$$

Roto-Translation Access Function A *roto-translation access function* maps a scalar equation index $e = (e_1, \ldots, e_n) \in E$ to a scalar variable index $v = (v_1, \ldots, v_m) \in V$ according to

$$f_{rt}(e) = \mathbf{A}e + \mathbf{b}$$

with $\mathbf{A}$ being a matrix of size $m \times n$, and $\mathbf{b}$ being a vector with m elements. The elements in matrix $\mathbf{A}$ are also constrained to the values in $\{0, 1\}$, and each row of $\mathbf{A}$ is mandated to have at most one non-zero element.

It should be noted that, when $n = m$, roto-translation accesses, as defined in MARCO, represent a subset of affine transformations, namely rotations, translations, and projections. However, due to the constraints on the matrix $\mathbf{A}$, additional affine transformations are excluded. For example, even though of interest in Modelica due to its support for strided loops, the scaling transformation remains unoptimized. This choice is motivated by the additional complexity required by the multidimensional data structures of Section 3.1, and, in ultimate analysis, by practical constraints of time and

effort. Nevertheless, the covered affine transformations encompass a broad range of relevant use cases – such as finite difference schemes on multidimensional grids – and already constitute a significant advancement over the generic access function implementation outlined at the beginning of this section.

Indeed, when considering a multidimensional range of points, the mapping operation can be performed on $O(d)$ time, with d being the number of dimensions, rather than needing a point-wise iteration over its points. Consider a multidimensional range of indices $\tilde{E} = \tilde{e_1} \times \ldots \times \tilde{e_n}$ and denote as $\tilde{V} = \tilde{v_1} \times \ldots \times \tilde{v_m}$ the mapped range that we are interested in computing. Define the points $l = (l_1, \ldots, l_n)$ and $u = (u_1, \ldots, u_n)$ such that $l_i = \tilde{e}_{iL}$ and $u_i = \tilde{e}_{iU}$. That is, l and u are the points respectively composed by the lower and upper bounds of the ranges of $\tilde{E}$. The mapping of $\tilde{E}$ through the roto-translation $\tilde{f_{rt}}$ can be computed as

$$\begin{aligned}\tilde{V} &= \tilde{f_{rt}}(\tilde{E}) \\ &= \tilde{v_1} \times \ldots \times \tilde{v_m} \mid\ \tilde{v}_{iL} = (\mathbf{A}l + \mathbf{b})_i, \\ &\qquad\qquad \tilde{v}_{iU} = \max\big((\mathbf{A}u + \mathbf{b})_i,\ \tilde{v}_{iL} + 1\big), \\ &\qquad\qquad \forall i = 1, \ldots, m\end{aligned}$$

The mapping $\hat{V} = \bar{f}(\hat{E})$, with $\hat{E}$ compactly implemented through an index set, can be efficiently obtained by iterating over the multidimensional ranges $\hat{\tilde{E}}_i$ composing $\hat{E}$, and by adding each correspetive mapped range $\hat{\tilde{V}}_i = \tilde{f}(\hat{\tilde{E}}_i)$ to the result index set using its `insert` operation (see Section 3.1).

The efficiency of the inverse mapping operation can also be improved. It is however necessary to distinguish between the case where $\mathbf{A}$ is an invertible square matrix, and the case where it is not.

In the first scenario, the inverse access function ${f_{rt}}^{-1}$ can be defined as

$${f_{rt}}^{-1}(v) = \mathbf{A}^{-1}(v - \mathbf{b})$$

where $\mathbf{A}^{-1}$ is the inverse of the matrix $\mathbf{A}$. The previous considerations can then be seamlessly applied to compute the inverse mapping of a multidimensional range or of an index set of variable indices.

In the second scenario, where $\mathbf{A}$ is not a square matrix or not invertible, a *reduced* inverse mapping is first computed as follows:

$$({f_{rt}}')^{-1}(v') = (\mathbf{A}')^{-1}(v' - \mathbf{b}')$$

where its components and argument are obtained as follows:

- The $\mathbf{A}'$ matrix is obtained by removing the zero-valued rows and columns from $\mathbf{A}$, thus obtaining an invertible matrix.
- The vector $\mathbf{b}'$ is obtained by removing the components that correspond to the zero-valued rows of $\mathbf{A}$.
- The point v' is obtained from v by removing the dimensions that correspond to the zero-valued rows of $\mathbf{A}$.

Given the reduced inverse mapping $\hat{E}'$ obtained by applying the inverse access function of the reduced problem, the full inverse mapping $\hat{E}$ can be obtained by reinserting the coordinates of the projected dimensions into the correct positions of each point $\hat{e}' \in \hat{E}'$.

Formally, let D be the set of all dimensions, and $D' \subseteq D$ the subset of dimensions retained in the reduced problem (i.e., the non-zero columns of $\mathbf{A}$). Then the inverse mapping $\hat{E}$ is computed as follows:

$$\begin{aligned}\hat{E} &= {f_{rt}}^{-1}(\hat{V}) \\ &= \bigcup_{\hat{e}' \in \hat{E}'} \bigcup_{e \in E} \{\hat{e} = (\hat{e_1}, \ldots, \hat{e_n}) \mid \hat{e}_i = \begin{cases} \hat{e}'_{i - \text{used}(i)} & \text{if } i \in D' \\ e_i & \text{if } i \notin D' \end{cases} \\ &\qquad\qquad \forall i = 1, \ldots, n\}\end{aligned}$$

with $\text{used}(i) = \text{card}(\{\ d : \ d < i \wedge d \notin D'\})$, which is, informally, the number of coordinates picked from e before reaching i while going from 1 to i.

Also in this case, the operation can be efficiently performed on a multidimensional range $\hat{\tilde{V}}$:

$$\begin{aligned}\hat{\tilde{E}} &= {f_{rt}}^{-1}(\hat{\tilde{V}}) \\ &= \bigcup_{\tilde{E}} \{\hat{\tilde{e}} = (\hat{\tilde{e_1}} \times \ldots \times \hat{\tilde{e_n}}) \mid \hat{\tilde{e}}_i = \begin{cases} ({\tilde{f'_{rt}}}^{-1}(\hat{\tilde{V}'}))_{i - \text{used}(i)} & \text{if } i \in D' \\ \tilde{e_i} & \text{if } i \notin D' \end{cases} \\ &\qquad\qquad \forall i = 1, \ldots, n\}\end{aligned}$$

where $\tilde{E} = \tilde{e_1} \times \ldots \times \tilde{e_n}$ represents a multidimensional range composing the index space E. Notice how $\hat{\tilde{E}}$ consists of a set of multidimensional ranges, rather than a single entity. This is due to the multiple possible ranges defining the index space E, each of which would see one or more its ranges as disappearing with the mapping operation.

An example of Roto-Translation Access Function For the sake of clarity, we provide an example of a roto-translation access function and compute its mapping and inverse mapping. Let us consider an equation $\mathcal{E}$ with a two-dimensional index space $E = [1, 3) \times [4, 6) \times [7, 9)$ (defined over the dimensions i_0, i_1, i_2), a variable $\mathcal{V}$ with index space $V = [1, 21) \times [1, 11)$, and the access function $(i_2 + 2, i_1 - 1)$. Table 3.1 shows the mapping and inverse mapping of the equation indices to the variable indices. Let us now consider the equation indices $\hat{E} = \{(1, 5, 7), (1, 5, 8), (2, 5, 7), (2, 5, 8)\} \subseteq E$, which can be compactly represented through the multidimensional range $\hat{\tilde{E}} = [1, 3) \times [5, 6) \times [7, 9)$, and let us compute its mapping, which should result in the multidimensional range $\hat{\tilde{V}} = [9, 11) \times [4, 5)$.

The matrix $\mathbf{A}$ and vector $\mathbf{b}$ of the roto-translation access function are defined as

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

Table 3.1: *Mapping and inverse mapping of equation indices to variable indices for the roto-translation access function* $(i_2 + 2, i_1 - 1)$.

Equation Index $e = (i_0, i_1, i_2)$			Variable Index $v = (v_0, v_1)$	
i_0	i_1	i_2	$v_0 = i_2 + 2$	$v_1 = i_1 - 1$
1	4	7	9	3
1	4	8	10	3
1	5	7	9	4
1	5	8	10	4
2	4	7	9	3
2	4	8	10	3
2	5	7	9	4
2	5	8	10	4

We compute $l = (1, 5, 7)$ and $u = (3, 6, 9)$, respectively made of the lower and upper bounds of the multidimensional range $\hat{\tilde{E}}$, and their mappings

$$\tilde{v_L} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 5 \\ 7 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 7 \\ 5 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 9 \\ 4 \end{bmatrix}$$

$$\tilde{v_U} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 9 \\ 6 \end{bmatrix} + \begin{bmatrix} 2 \\ -1 \end{bmatrix} = \begin{bmatrix} 11 \\ 5 \end{bmatrix}$$

The resulting mapped multidimensional range is therefore

$$\hat{\tilde{V}} = [\tilde{v_{L1}}, \tilde{v_{U1}}) \times [\tilde{v_{L2}}, \tilde{v_{U2}}) = [9, 11) \times [4, 5)$$

which is indeed the expected result.

Let us now start from $\hat{\tilde{V}}$, and go back to the original equation indices $\hat{\tilde{E}}$. The matrix $\mathbf{A}$ is a 2×3 matrix and, due to its non-invertible nature, we compute the reduced $\mathbf{A}'$, $(\mathbf{A}')^{-1}$, and $\mathbf{b}'$ entities:

$$\mathbf{A}' = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad (\mathbf{A}')^{-1} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{b}' = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

All the rows of $\mathbf{A}$ are non-zero (i.e., all the components of $\hat{\tilde{V}}$ are dimension-dependent), thus $\hat{\tilde{V}}' = \hat{\tilde{V}}$. $\hat{\tilde{E}}'$ is computed as

$$\tilde{e_L} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 9 \\ 4 \end{bmatrix} - \begin{bmatrix} 2 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 7 \\ 5 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \end{bmatrix}$$

$$\tilde{e_U} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 11 \\ 5 \end{bmatrix} - \begin{bmatrix} 2 \\ -1 \end{bmatrix} \right) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 9 \\ 6 \end{bmatrix} = \begin{bmatrix} 6 \\ 9 \end{bmatrix}$$

$$\hat{\tilde{E}}' = \{\hat{\tilde{e}} = [\tilde{e_{L1}}, \tilde{e_{U1}}) \times [\tilde{e_{L2}}, \tilde{e_{U2}})\} = \{[5, 6) \times [7, 9)\}$$

To reinsert the projected dimensions, let us first compute the values of used$(i), \forall\ 1 \leq i \leq n$, with $D = \{1, 2, 3\}$ and $D' = \{2, 3\}$:

$$\text{used}(1) = 0, \quad \text{used}(2) = 1, \quad \text{used}(3) = 1$$

By reinserting the ranges of the projected dimensions, we obtain the final inverse mapping

$$\hat{\tilde{E}} = \{\tilde{e}_1 \times [\tilde{e_{L1}}, \tilde{e_{U1}}) \times [\tilde{e_{L2}}, \tilde{e_{U2}})\} = \{[1, 3) \times [5, 6) \times [7, 9)\}$$

Composition of Access Functions Before proceeding with further analyses and optimizations, we provide a short but relevant definition for the operation of *composition* of two access functions. Given two access functions $f_1 : \mathbb{Z}^n \to \mathbb{Z}^m$ and $f_2 : \mathbb{Z}^m \to \mathbb{Z}^p$, the composition $f = f_1 \circ f_2 : \mathbb{Z}^n \to \mathbb{Z}^p$ is defined as

$$f(x) = f_2(f_1(x))$$

As demonstrated in the following sections, the composition of access functions, combined with their inversion, is a powerful tool for reasoning on the relationships between equations without having to necessarily resort to scalar analyses.

In the case of two roto-translation access functions $f_1(x) = \mathbf{A_1}x + \mathbf{b_1}$ and $f_2(x) = \mathbf{A_2}x + \mathbf{b_2}$, with well-posed dimensionalities, the composition can be computed as follows:

$$\begin{aligned} f(x) &= f_1(x) \circ f_2(x) = f_2(f_1(x)) \\ &= \mathbf{A_2}(\mathbf{A_1}x + \mathbf{b_1}) + \mathbf{b_2} = (\mathbf{A_2}\mathbf{A_1})x + (\mathbf{A_2}\mathbf{b_1} + \mathbf{b_2}) \end{aligned}$$

3.3 Array-Aware Causalization

As anticipated in Section 2.3, the causalization of a DAE system is a multi-step process that involves multiple analyses. In this section, we focus on the array-aware algorithms and data structures that MARCO implements to perform the causalization of array-oriented DAE systems.

3.3.1 Data Structures for Array-Aware Matching

Previous research by Fioravanti et al. [26] introduced a heuristic algorithm capable of performing array-aware matching of array-oriented DAE systems in polynomial time. However, despite such a groundbreaking advance in the field, some questions remained open: the implementation of the so-called *local matching solution* was not fully addressed, and, despite not impacting on the overall complexity of the algorithm, an efficient implementation remains crucial to achieve an optimally-close matching. Moreover, regarding the underlying data structures, the proposed Multidimensional Compressed Incidence Matrix (MCIM) representation was later found to be inefficient for access patterns going beyond pure translations. Such inefficiency was briefly showcased by Agosta et al. [4] during the experimental evaluation campaign, where the

same model that will be re-evaluated in Chapter 7, characterized by a regular structure, exhibited a non-constant compilation time.

In this section, we build upon the foundations laid in the previous sections, where we introduced the necessary data structures for representing points in an arbitrarily high-dimensional space and for mapping the relationship between equations and variables. We now present an improved design for the MCIM data structure, and see how the local matching solutions can be immediately obtained out of that new implementation.

A New Design for the MCIM Given an equation $\mathcal{E}$ and a variable $\mathcal{V}$, respectively with indices E and V enumerable through i and j, the incidence matrix $\mathbf{M}$ is a $|E| \times |V|$ matrix such that $\mathbf{M}_{ij} = 1$ if there exists an access function f between E and V such that (i, j) belongs to the relation f, and 0 otherwise. More informally, an incidence matrix keeps track of all the access functions between a given array equation and array variable, and represents the relationship between their scalar indices in a compact way.

In the initial proposal by Fioravanti et al. [26], the MCIM was designed as a set of paired objects, with the first element representing a multidimensional range of equation indices, and the latter representing a list of offsets to be applied to the dimensions of the range. Such a design perfectly fits a pure affine translation, where the $\mathbf{A}$ matrix is an identity matrix, but struggles in all other circumstances. For example, in case of constant access functions, the offset between equation and variable indices is not constant, thus forcing the creation of a new pair for each equation index. In other words, the initial design forces the interpretation of any access function as an affine translation, which, of course, is not always the case, and was indeed later found to be the cause of the non-constant scaling behavior observed in Agosta et al. [4].

The improved MCIM data structure is built around the concept of *groups*. Each group encapsulates a subset of equation indices (the *keys*) and associates them with a unique access function that maps these equation indices to the corresponding variable indices (the *values*). This design generalizes the previous offset-based approach, allowing the MCIM to efficiently represent arbitrary access patterns, including translations, constant accesses, and more complex relationships.

A group G within the MCIM is defined as a tuple (K, f), where K is a set of equation indices and f is an access function mapping each equation index in K to a variable index in V. The MCIM data structure is then represented as a set of such groups, coupled with an index set I defined over a multidimensional space of cardinality $|E| + |V|$. The index set I treats every point of the matrix as a unique point composed by the concatenation of equation and variable indices, and it is used for arbitrary modifications of the incidence matrix not based on any access function.

The generic point p of the matrix is defined as $p = \langle p_e, p_v \rangle$, where $p_e = (p_{e1}, \ldots, p_{en}) \in E$ and $p_v = (p_{v1}, \ldots, p_{vm}) \in V$ (with $n = |E|$ and $m = |V|$) respectively indicate the row and the column of the equation and of the variable in the hyperdimensional incidence matrix.

A MCIM M can therefore be formally defined as a pair

$$M = (\mathcal{G}, I)$$

where the following definitions and constraints hold:

- $\mathcal{G} = \{(K, f) \mid K \subseteq E, f : K \to V\}$

- I is an index set defined over a multidimensional space of cardinality $|E| + |V|$ and it holds that

$$i = \langle i_E, i_V \rangle = (i_{E1}, \ldots, i_{En}, i_{V1}, \ldots, i_{Vm}) \in I \implies i_E \in E \wedge i_V \in V$$

Supported Operations The efficiency of a MCIM relies on the usage of the groups as much as possible, thus avoiding the use of the index set I whenever feasible. The insertion of multiple points can be performed by providing their equation coordinates and an access function relating them to the variable coordinates. Such an insertion is performed by checking if the access function is already present in any of the existing groups, and if so, by adding the equation indices to the keys of that group. If not, a new group is created with the provided equation indices and access function. More formally, given a MCIM $M = (\mathcal{G}, I)$, the insertion of equation indices K with access function f is performed as follows:

$$M \cup (K, f) = \begin{cases} (\mathcal{G} \cup \{(K, f)\}, I) & \text{if } \nexists\, (K_G, f_G) \in \mathcal{G} : f = f_G \\ (\mathcal{G}', I) & \text{if } \exists\, (K_G, f_G) \in \mathcal{G} : f = f_G \end{cases}$$

$$\text{with} \quad \mathcal{G}' = (\mathcal{G} \setminus \{(K_G, f_G) \mid (K_G, f_G) \in \mathcal{G} \wedge f_G = f\}) \cup \{(K \cup (\bigcup_{(K_G, f_G) \in \mathcal{G} \mid f_G = f} K_G), f)\}$$

Operations involving individual points are also possible. Given a point $p = \langle p_e, p_v \rangle$, we define the containment operator $\in$ for a group $G = (K, f)$ as

$$p \in G \Leftrightarrow (p_e \in K) \wedge (p_v \in f(p_e))$$

and for a MCIM $M = (\mathcal{G}, I)$ as

$$p \in M \Leftrightarrow (p \in I) \vee (\exists\, G \in \mathcal{G} : p \in G)$$

For convenience, we also introduce the operator $\tilde{\in}$ to check if a point $p = \langle p_e, p_v \rangle$ can be contained by a group $G = (K, f)$:

$$p \,\tilde{\in}\, G \Leftrightarrow f(p_e) = \{p_v\}$$

The insertion of a point p into a MCIM $M = (\mathcal{G}, I)$ is performed by checking if p can be contained by any of the existing groups in $\mathcal{G}$ and, if not, by inserting it into the index set I:

$$M \cup p = \begin{cases} (\mathcal{G}, I \cup \{p\}) & \text{if } \nexists\, G \in \mathcal{G} : p \,\tilde{\in}\, G \\ (\mathcal{G}', I) & \text{if } \exists\, G \in \mathcal{G} : p \,\tilde{\in}\, G \end{cases}$$

with $\mathcal{G}' = (\mathcal{G} \setminus \{G \mid G \in \mathcal{G} \wedge p \,\tilde{\in}\, G\}) \cup \{G' \mid G' = (K_G \cup \{p_e\}, f_G) \wedge (K_G, f_G) \in \mathcal{G} \wedge p \,\tilde{\in}\, (K_G, f_G)\}$.

The insertion is similar in the case of a set of points (e.g., represented through a range or index set) paired with an access function: a group having the same access function is searched for, and if found, the points are added to its keys; otherwise, a new group is created with the provided points and access function. The insertion of

points to the keys benefits from the efficient data structures and algorithms described in Section 3.1, thus allowing for efficient handling of large sets of points.

Other operations are also possible, like the removal of a point or range of points, the union of two MCIMs, their subtraction, and the projection of rows or columns. However, for the sake of brevity we do not detail these operations here, and refer the reader to the actual implementation of the compiler.

Local Matching Solutions An important aspect that is worth addressing, instead, consists in the ability to extract local matching solutions from the MCIM structure. This task is performed by iterating over all the groups in the MCIM and by applying a procedure that is customized according to the nature of its access function. The set of local matching solutions is defined as the union of the local matching solutions of each group, and by a group for each point of the sparse index set (for which, unfortunately, no optimization is possible).

In MARCO, an optimized procedure is implemented in the case of roto-translation access functions. Given a group $G = (K, f)$, in which f is a roto-translation, the local matching solution is computed as follows.

If f does not project any dimension (i.e., all dimensions are used), the local matching solution consists of a MCIM $M' = (\mathcal{G}', I')$ where $\mathcal{G}'$ contains a single group $G' = (K, f)$.

Instead, if some dimensions $(d_i, \ldots, d_k)$ are unused, it means that the same local matching solution will be repeated for every possible value in $d_i \times \ldots \times d_k$. In other words, projected dimensions would have no influence on the mapping, and therefore every column would have either zero or $d_i \times \ldots \times d_k$ one-valued cells. To obtain valid local matching solutions, where each row and column of the incidence matrix has at most one 1, we need to create a separate group for each combination of the unused coordinates' possible values, thus leading to $|d_i| \cdot \ldots \cdot |d_k|$ groups in $\mathcal{G}'$. Each group $G' = (K', f)$ in $\mathcal{G}'$ is obtained by starting from the original keys K, and by replacing the ranges corresponding to the unused dimensions each time with a different combination of their possible values.

In a more formal way, let us consider a group $G = (K, f)$, where K can be described as a set of multidimensional ranges $\tilde{K} = \{\tilde{K}_i\}$, with $\tilde{K}_i = \tilde{k}_{i1} \times \ldots \times \tilde{k}_{in}$, and denote as P the set of projected dimensions. The set of local matching solutions $\mathcal{M}' = \{(G'_i, \emptyset)\}$, with $1 \leq i \leq |\tilde{K}|$, for the group G is such that

$$G'_i = \{(K', f) \mid \; K' = \tilde{e}_1 \times \ldots \times \tilde{e}_n, \tilde{e}_d = \begin{cases} [p, p+1), p \in \tilde{k}_{id} & \text{if } d \in P \\ \tilde{k}_{id} & \text{if } d \notin P \end{cases} \}$$

3.3.2 Computation of Strongly-Connected Components

The identification of SCCs is a fundamental step to obtain executable code, and finds an initial array-oriented formulation in the master thesis of Fioravanti [25]. The author demonstrated that the recognition of SCCs can be significantly accelerated by exploiting the property that if two variables are not part of the same SCC at the array level, they surely will not be at the scalar level either. However, despite enabling a more efficient pruning of the search space, this property does not, by itself, provide a complete

array-aware characterization of SCCs, as scalar-level analysis remains necessary for array-level candidate SCCs. In this section, we detail a four-phased algorithm implemented in MARCO to fill this gap.

Coarse-grained SCCs Computation In the first phase, the proposed solution constructs an *array dependency graph* G_1 to identify SCCs at the array level. In this graph, each node $\mathcal{E}$ corresponds to an array equation, characterized by its index space E. Directed edges are established from node $\mathcal{E}_i$ to node $\mathcal{E}_j$ (with i possibly equal to j) whenever $\mathcal{E}_i$, for any index $e \in E$, reads a scalar variable that is written by $\mathcal{E}_j$.

To detect the array-level SCCs, we apply Tarjan's algorithm to G_1, resulting in a collection of strongly-connected components that we denote as $\mathcal{S}_1$. This phase builds upon previous work by Fioravanti [25], but extends it by explicitly incorporating access functions into the construction of the dependency graph, thereby enabling a more precise characterization of dependencies at the array level.

Per-SCC Cycles Computation In the second phase, the algorithm refines the analysis by focusing on each array-level SCC $\mathcal{S}_{1i} \in \mathcal{S}_1$ individually. For every equation $\mathcal{E}$ within $\mathcal{S}_{1i}$, a Depth-First Search (DFS) traversal is performed on the subgraph G_{2i}, which is constructed by restricting G_1 to the nodes and edges belonging to $\mathcal{S}_{1i}$. These traversals identify all cycles present in G_{2i}, while precisely tracking the specific indices of each equation that participates in these cycles.

Before presenting the detailed steps of the algorithm, we introduce the notation and definitions that will be used throughout its description:

- Let $\mathcal{E}$ be an equation in the SCC, with index space E.
- Let $visited : \mathcal{E} \mapsto S_E$ be a map between a generic equation $\mathcal{E}$ and a set $S_E \subseteq E$ representing the indices of $\mathcal{E}$ which have already been visited during the application of the algorithm.
- Let a cycle be defined as a list of tuples $(\mathcal{E}, \hat{E}, \mathcal{V}_W, V_W, f_W, \mathcal{V}_R, V_R, f_R)$ where $\mathcal{E}$ is an equation participating in the cycle, $\hat{E} \subseteq E$ is the set of indices of $\mathcal{E}$ involved in the cycle, $\mathcal{V}_W$ is the variable written by $\mathcal{E}$, V_W is the set of indices of $\mathcal{V}_W$ written by $\mathcal{E}$, f_W is the access function that writes to $\mathcal{V}_W$, $\mathcal{V}_R$ is a variable read by $\mathcal{E}$, V_R is the set of indices of $\mathcal{V}_R$ read by $\mathcal{E}$, and f_R is the access function that reads from $\mathcal{V}_R$.
- Let $V_{\mathcal{E}\mathcal{V}}$ be the set of indices of variable $\mathcal{V}$ that are matched to (or, equivalently, written by) equation $\mathcal{E}$.
- An access function f of $\mathcal{E}$ w.r.t. variable $\mathcal{V}$ is a *write* access function for a set of given indices $\hat{E} \subseteq E$ if $V_{\mathcal{E}\mathcal{V}} \subseteq f(\hat{E})$. It is, instead, a *read* access function if $V_{\mathcal{E}\mathcal{V}} \nsubseteq f(\hat{E})$.

The algorithm is applied to each equation $\mathcal{E} \in \mathcal{S}_{1i}$, and operates recursively. The set of recognized cycles is denoted as $\mathcal{C}$, and it is initially empty. The *path* being explored is an empty list of tuples in the form $(\mathcal{E}, \hat{E}, \mathcal{V}_W, V_W, f_W, \mathcal{V}_R, V_R, f_R)$ where $\mathcal{E}$, and it is denoted as P. For each equation $\mathcal{E}$, the first invocation is performed with $\hat{\mathcal{E}} = \mathcal{E}$, $\hat{E} = E$, and the current path P being an empty list.

1. Collect all the write access functions f_{Wi} in $\hat{\mathcal{E}}$ with respect to index space $\hat{E}$. For each such function f_{Wi} accessing variable $\mathcal{V}_W$:
   (a) Initialize the set R with all the access functions f_R obeying the former definition of read access function for the index space $\hat{E}$.
   (b) For all write access functions f_{Wj} with $j \neq i$, check if $f_{Wj}(\hat{E}) \setminus f_{Wi}(\hat{E}) \neq \emptyset$. If so, add f_{Wj} to the set R.
   (c) For each $f_R \in R$ accessing variable $\mathcal{V}_R$:
      i. Compute the set of read indices $V_R = f_R(\hat{E})$. If f_{Wi} and f refer to the same variable, then remove the indices $f_{Wi}(\hat{E})$ from V_R.
      ii. For each equation $\mathcal{E}'$ (even partially) matched to the indices V_R of variable $\mathcal{V}_R$, compute the inverse mapping $\hat{E}_R$ of $V_R \cap V_{\mathcal{E}'\mathcal{V}}$ through the write access functions of $\mathcal{E}'$ (i.e., the union of the indices given by all inverse mappings). Define $P' = P$ and append a new tuple $(\mathcal{E}, \hat{E}, \mathcal{V}_W, f_{Wi}(\hat{E}), f_{Wi}, \mathcal{V}_R, V_R \cap V_{\mathcal{E}'\mathcal{V}}, f_R)$ to P'. For convenience, also denote $\hat{E}_R$ as $\hat{E}'$.
      iii. Restrict the indices of P', so that for each pair of consecutive tuples $(\mathcal{E}_i, \hat{E}_i, \mathcal{V}_{Wi}, V_{Wi}, f_{Wi}, \mathcal{V}_{Ri}, V_{Ri}, f_{Ri})$ and $(\mathcal{E}_j, \hat{E}_j, \mathcal{V}_{Wj}, V_{Wj}, f_{Wj}, \mathcal{V}_{Rj}, V_{Rj}, f_{Rj})$ in P', it holds that $V_{\mathcal{R}i} = V_{\mathcal{W}j}$, $\hat{E}_i = {f_{Ri}}^{-1}(V_{\mathcal{R}i})$, and $V_{\mathcal{W}i} = f_{Wi}(\hat{E}_i)$.
      iv. Check if there exists a tuple $(\mathcal{E}_i, \hat{E}_i, \mathcal{V}_{Wi}, V_{Wi}, f_{Wi}, \mathcal{V}_{Ri}, V_{Ri}, f_{Ri})$ in P' such that $\mathcal{E}_i = \mathcal{E}' \wedge \hat{E}_i \cap \hat{E}_R$. If so, add to $\mathcal{C}$ the cycle obtained by extracting from P' all the tuples from the first position to the one satisfying the condition.
      v. Recursively apply the algorithm (by starting from point (1)) to each equation $\mathcal{E}'$ for which a new tuple has been added in point (ii). The index set $\hat{E}$ for the recursive call is set to $\hat{E}'$, while P is set to P'.

Cycles Partitioning The third phase of the algorithm refines the set of cycles by partitioning the indices of each equation involved. Specifically, for every equation that appears in multiple cycles, its index space is split such that no two distinct cycles share overlapping indices for the same equation. This ensures that each cycle is associated with a unique, non-intersecting subset of indices for every participating equation. Formally, the new set of cycles is such that $\nexists\, i, j \mid i \neq j \wedge \mathcal{E}_i = \mathcal{E}_j \wedge \hat{E}_i \cap \hat{E}_j \neq \emptyset \wedge \hat{E}_i \neq \hat{E}_j$.

The implementation of this phase is straightforward, and therefore we do provide only a high-level description. The algorithm starts with an empty set of cycles $\mathcal{C}'$. For every cycle $C \in \mathcal{C}$, the algorithm first splits all the cycles already inserted into $\mathcal{C}'$ which contain an equation that partially overlaps with the current cycle C (i.e., the two sets of indices overlap but are not equal). The cycle C is then also split to remove the partial overlap, and the resulting cycles are inserted into $\mathcal{C}'$. In splitting the cycles, the resulting new cycles are updated to ensure that all the reported indices of written and read variables remain consistent with the indices of the equation they are associated with.

Fine-Grained SCCs Computation In the fourth phase, the algorithm constructs an array dependency graph G_3, leveraging the index partitions established in the previous step.

Unlike G_1, the strongly-connected components that can be identified in G_3 are now composed of equations whose associated indices are strictly disjoint from those of any other SCC.

3.3.3 Equations Sorting

The last step of the causalization pipeline consists of sorting the equations in such a way that the dependencies between variables are respected during the execution of the simulation. Naturally, in the presence of SCCs, the equations within a SCC can not be ordered with respect to each other due to their cyclic nature, and the whole SCC is therefore considered as a single entity to be executed before or after other equations.

Consider the case of scalar representations, where no array variable or array equation is defined. In this scenario, it is possible to build a Directed Acyclic Graph (DAG) whose nodes are the union of all SCC and the equations not participating in any cyclic relation. For the sake of simplicity, both kinds of nodes are considered as SCC, defining the latter as a *trivial* SCC. The directed edges of the DAG are defined by the read and write dependencies between the equations, and an edge from node S_i to node S_j is inserted if

$$\exists\, e_i \in S_i, e_j \in S_j : e_i \text{ reads } \mathcal{V} \wedge e_j \text{ writes } \mathcal{V}$$

where $\mathcal{V}$ is a variable of the DAE system. In the scalar case, the DAG is guaranteed to be acyclic by construction, as all cycles are indeed already collapsed into their own nodes. A topological sort of the DAG can then be performed, yielding a linear order of the SCCs that respects the dependencies between them.

In the case of array equations, the situation is more complex. Given two SCCs S_i and S_j, an edge from S_i to S_j is inserted in the dependency graph if, given a variable $\mathcal{V}$ defined over indices V, it holds that

$$\exists\, e_i \in S_i, e_j \in S_j : e_i \text{ reads } (\mathcal{V}, \hat{V}_i) \wedge e_j \text{ writes } (\mathcal{V}, \hat{V}_j) \wedge V_i \cap V_j \neq \emptyset$$

where $\hat{V}_i \subseteq V$ and $\hat{V}_j \subseteq V$.

In the array case, the dependency graph is not guaranteed to be acyclic, as the range of indices read and written by a certain array equation may overlap, leading to a self-loop in the graph. However, a topological sort of the graph can still be performed to obtain a linear order between the SCCs, and later analyze each self-dependent trivial SCC to understand if a cyclic dependency would arise on a scalar level.

Given a topological sort $T = [S_1, \ldots, S_n]$ of the array dependency graph G, with each S_i being a SCC of G, the final sorting of equations T' can be obtained as follows. Each element $S_i \in T$ is processed in order. If S_i is a non-trivial SCC, or a trivial SCC with no self-loop, then it is added to T': the array equations in S_i can be kept in their compact representation, and the order of visit of their indices is irrelevant. If, instead, S_i is a trivial SCC with a self-loop, then the only array equation in S_i is expanded into a set of scalar equations (i.e., each defined over a single point of the original index space), and both the detection of SCCs and their topological sorting are performed on the graph built out of those scalar equations. This last topological sort is then appended to T'.

Of course, the proposed handling of self-loops is not ideal and calls for further improvements. The array nature of an equation should ideally be preserved, avoiding

the need for later reconstruction. In particular circumstances, this can be achieved by analyzing the structure of the involved access functions and, in the remainder of this section, we detail the recognition of one of such cases. Even though this will not be a general solution, it should be noted that compilers rarely provide one-fits-all solutions, but rather often rely on the recognition of specific patterns for which optimizations are known to be applicable. Similar optimizations may therefore be devised in the future to improve a wider range of models.

Optimizing Roto-Translations Consider an array equation $\mathcal{E}$ defined over an index space E with n dimensions. The goal is to determine whether the loops over which the equation is declared may be preserved, and to determine the n iteration directions of such loops (i.e., from the lower to the upper bound, or the opposite).

Let f_W be one write access of E (as defined in Section 3.3.2) such that f_W is an invertible roto-translation access (i.e., its $\mathbf{A}$ matrix is not singular). If such f_W exists, we then compute the set of access functions $F_\circ = \{f_\circ \mid f_\circ = {f_W}^{-1} \circ f_R\}$, with f_R being a read access function accessing the same variable as f_W with overlapping indices (i.e., $f_W(E) \cap f_R(E) \neq \emptyset$).

If all access functions $f_\circ$ in $F_\circ$ are pure translations (i.e., roto-translations with $\mathbf{A}$ being an identity matrix), then the analysis may proceed. For each row i of $\mathbf{A}_{{f_W}^{-1}}$, determine j such that column j is the only 1-valued column along row i. This column is unique by construction. We then characterize each dimension as follows:

- If the j-th element of all vectors $\mathbf{b}_{f_\circ}$ is zero, then the iteration direction of the i-th dimension can be chosen arbitrarily.
- If the j-th element of all vectors $\mathbf{b}_{f_\circ}$ is zero or negative, then the iteration on the i-th dimension must go from its lower bound to its upper bound. We call this a *forward* iteration.
- If the j-th element of all vectors $\mathbf{b}_{f_\circ}$ is zero or positive, then the iteration on the i-th dimension must go from its upper bound to its lower bound. We call this a *backward* iteration.

If, for any i, none of the above conditions hold, the optimization is not applicable, and the array equation should be expanded into scalar equivalents.

Informally, the optimization operates by bringing the write access into identity form by applying the ${f_W}^{-1}$ function to the whole equation, and the f_W function to the iteration space. The composition ${f_W}^{-1} \circ f_R$ permits to determine whether, in the remapped iteration space, the read indices numerically come before or after the written ones (e.g., $[i-1]$ entails a dependency on the element written at the previous iteration i). The multiplication by the matrix $\mathbf{A}_{{f_W}^{-1}}$ is then used to map the chosen iteration directions back to the original dimensions, which may have undergone a rotation when applying f_W to the index space of the equation.

An example Consider the following equation, defined over the index space $E = [3, 6) \times [9, 11)$:

$$x[i_1 + 1, i_0 - 2] = x[i_1, i_0 - 1]$$

with $x[i_1 + 1, i_0 - 2]$ being the write access function f_W and $x[i_1, i_0 - 1]$ being the read access function f_R. From an array perspective, the written indices are $[10, 12) \times [1, 4)$, and the read indices are $[9, 11) \times [2, 5)$. The two ranges overlap on indices $[10, 11) \times [2, 4)$, and therefore the array equation manifests a self-loop in the dependency graph. From a scalar perspective, however, the equation unrolls as shown in Table 3.2, and no cyclic dependency arises between the resulting scalar equations.

i_0	i_1	Equation
3	9	$x[10, 1] = x[9, 2]$
3	10	$x[11, 1] = x[10, 2]$
4	9	$x[10, 2] = x[9, 3]$
4	10	$x[11, 2] = x[10, 3]$
5	9	$x[10, 3] = x[9, 4]$
5	10	$x[11, 3] = x[10, 4]$

Table 3.2: *Scalar expansion of the array equation* $x[i_1+1, i_0-2] = x[i_1, i_0-1]$ *over* $E = [3, 6)\times[9, 11)$.

To determine the absence of scalar cycles without unrolling the equation, the optimization described above may be applied on the array equation and its access functions. f_W is indeed an invertible roto-translation, and can be composed with f_R to obtain $f_\circ$:

$$f_W(x) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$f_W{}^{-1}(x) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

$$f_R(x) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\begin{aligned} f_\circ(x) = f_W{}^{-1}(x) \circ f_R(x) &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \end{aligned}$$

The access function $f_\circ$ is a pure translation, and its $\mathbf{b}$ vector indicates that dimensions i'_0, i'_1 of the remapped iteration space ($[10, 12) \times [1, 4)$) should be respectively iterated in forward and backward directions. After applying the rotation entailed by $f_W{}^{-1}$, the iteration directions can be mapped back to the original dimensions i_0 and i_1, which should therefore be respectively iterated in backward (from 5 to 3) and forward (from 9 to 10) directions. On a scalar level, such iteration orders would lead to the following list of assignments, in which data dependencies are respected:

1. $i_0 = 5, \quad i_1 = 9 : x[10, 3] \leftarrow x[9, 4]$
2. $i_0 = 5, \quad i_1 = 10 : x[11, 3] \leftarrow x[10, 4]$
3. $i_0 = 4, \quad i_1 = 9 : x[10, 2] \leftarrow x[9, 3]$
4. $i_0 = 4, \quad i_1 = 10 : x[11, 2] \leftarrow x[10, 3]$

5. $i_0 = 3, \quad i_1 = 9 : x[10, 1] \leftarrow x[9, 2]$
6. $i_0 = 3, \quad i_1 = 10 : x[11, 1] \leftarrow x[10, 2]$

CHAPTER 4

MLIR Dialects for DAE Systems

This chapter describes how the MLIR framework can be used to overcome the limitations of state-of-the-art design of Modelica compilers. More in detail, this chapter details the MLIR dialects that have been designed to compile a Base Modelica model within MARCO. Following the multi-layered approach of MLIR, these dialects aim to preserve the high-level concepts as much as possible while moving towards the low-level LLVM-IR code, as well as to conceptually separate the different aspects of the compilation process and interactions with the runtime environment.

4.1 A Multi-Level Approach to Compilation

Compiler technology has undergone a profound transformation over the past decades, driven by the increasing complexity of programming languages and the demand for high-performance code generation. Modern compilers are no longer monolithic programs tailored to a single language or target architecture. Instead, they are built as modular frameworks that can be extended and adapted to different source languages [7].

A typical compiler is organized into three main stages: the *frontend*, the *middleend*, and the *backend*. The frontend is responsible for parsing the source code, performing lexical and syntactic analyses, and generating an IR that captures the semantics of the input program while being independent from the source language itself. The middleend operates on this language-agnostic IR, applying a series of optimizations that improve performance, reduce resource usage, and enable advanced features such as parallelism and vectorization. Finally, the backend performs target-specific optimizations and translates the IR into executable code, such as machine instructions or bytecode. This separation of concerns allows compiler developers to focus on specific aspects of the compilation process and facilitates the reuse of components across

different languages and targets.

Over the past twenty years, the LLVM project [42] has become a prominent framework for modular compiler design, providing a comprehensive set of tools and libraries for building compilers. The language-agnostic IR used by LLVM – named *LLVM-IR* – is designed to be independent from any specific programming language or architecture, while remaining human-readable. An example of LLVM-IR is shown in Listing 4.1, and can represent the result of compiling the Modelica function shown in Listing 4.2. The use of *can* here is deliberate, emphasizing that the LLVM-IR representation is not

```
1 define i32 @sum(i32 %a, i32 %b) {
2   %result = add i32 %a, %b
3   ret i32 %result
4 }
```

Listing 4.1: *Example of LLVM-IR code that computes the sum of two integers.*

```
1 function sum
2   input Integer a;
3   input Integer b;
4   output Integer result;
5 algorithm
6   result := a + b;
7 end sum;
```

Listing 4.2: *Modelica function that can be compiled to the LLVM-IR shown in Listing 4.1.*

uniquely determined by the Modelica source code. Because Modelica operates at a high level of abstraction and does not enforce specific implementation details such as integer bit widths, the compiler has the flexibility to make different lowering choices. For instance, while Listing 4.1 uses 32-bit integers (`i32`), the compiler could equally well target 64-bit integers (`i64`) or other representations based on platform requirements or optimization strategies. While this kind of representational ambiguity exists even for mainstream programming languages such as C, it is further stressed in Modelica due to the presence of higher-level abstractions that do not typically belong to the programming domain, such as DAEs. These higher-level abstractions are often lost – or, at the very least, become harder to represent – during the lowering process to LLVM-IR, whose semantics remain close to the C language, and the implementation of the compiler inevitably becomes more complex to bridge this semantic gap.

Motivated by the need for more flexible and extensible IRs, the compiler community has developed the Multi-Level Intermediate Representation (MLIR) framework [43], which represents a significant advancement in compiler infrastructure. MLIR provides a modular and extensible architecture that supports multiple levels of abstraction throughout a compiler's pipeline. Unlike LLVM-IR, which operates at a fixed level of abstraction close to machine code, MLIR enables the definition of custom IRs – called *dialects* – that can capture domain-specific semantics at various levels of abstraction. This multi-level approach allows compilers to preserve high-level information during the compilation process, enabling more sophisticated analyses and optimizations.

MLIR dialects are collections of operations, types, and attributes that define the semantics of a particular language or domain. For example, the `tensor` dialect can represent high-level mathematical operations on multidimensional arrays focusing only on value semantics, thus entirely abstracting the storage model that is instead introduced later in the compilation pipeline. The `affine` dialect introduces polyhedral representations for loop nests and data access patterns. The `cf` dialect represents control flow constructs, and the `scf` presents more structured programming concepts like loops and

conditionals. These dialects, and many others, can be composed and progressively lowered through a series of transformation passes, gradually reducing the level of abstraction until reaching target-specific representations.

The framework provides a flexible infrastructure for defining custom parsing and printing routines, verification logic, and transformation passes that can also operate on an interface-based manner. This design makes MLIR particularly well-suited for Domain-Specific Languages (DSLs) like Modelica, where domain-specific constructs such as equations, components, and physical connections require specialized representations and optimization strategies that cannot be adequately captured by LLVM-IR or similar solutions.

4.2 Bringing MLIR to the Modelica Environment

An initial MLIR-based prototype of the MARCO compiler was developed in 2021 by the same author of this thesis [62], and demonstrated promising results in terms of extensibility and optimization [4]. Building on this foundation, this manuscript proposes a more in-depth adoption of the MLIR framework, including the introduction of additional dialects that cover integration with external solvers and an improved coupling with the runtime environment, which no longer serves only as a collection of utility functions but now represents a core part of simulation control and hardware exploitation.

The MLIR-oriented compilation flow of MARCO is summarized in Figure 4.1. The frontend of MARCO accepts both Modelica and Base Modelica code. In the case of Modelica code, the source is first translated into Base Modelica using the frontend component of the OpenModelica compiler. It should be noted that this offloading decision has been driven by time and effort constraints, and, in principle, the frontend of MARCO can also be extended to perform the conversion – that is, *model instantiation* and *flattening* – by itself.

The Abstract Syntax Tree (AST) corresponding to the Base Modelica code is then represented as a newly introduced MLIR dialect named `bmodelica`, and several analyses and transformations (e.g., causalization, automatic differentiation) are applied, optionally bringing other new dialects into the IR (e.g., for interfacing with external solvers).

Once the mathematical aspects of the model have been addressed, the IR is converted to a set of core MLIR dialects, such as the `arith` and `func` ones, on which upstream transformations – not bound to the semantics of Modelica – can be applied. These transformations include optimizations such as bufferization, loop fusion, and others, which are common in many compiler toolchains. The resulting IR is then converted into LLVM-IR, so that it can be finally compiled by the backend of LLVM and linked with the libraries providing the runtime environment.

The most relevant components of the newly introduced MLIR dialects are examined in the remainder of this chapter, while the modifications to the IR, together with additional operations that are not essential to this initial overview, are presented in Chapter 5. It is acknowledged that the rationale underlying certain design choices may not be immediately apparent; however, the proposed design has been developed with careful consideration of the overall architecture of the toolchain. Each component is intended to function as an integral part of the system, rather than operating in isolation, and a holistic approach is therefore required to properly understand the upcoming

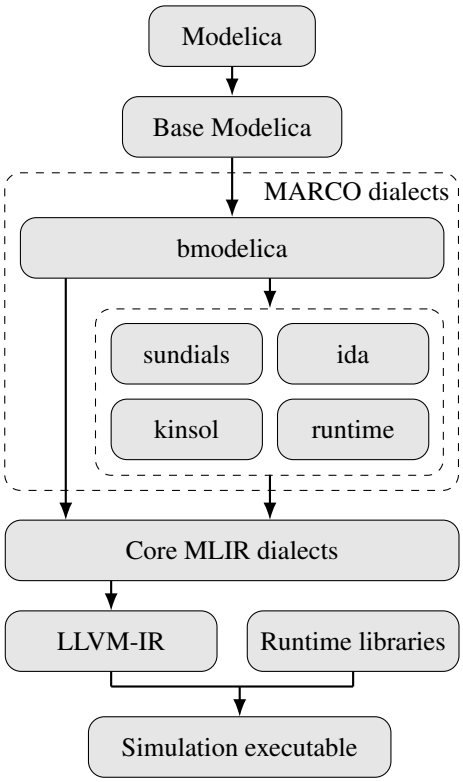

Figure 4.1: *Overview of the MARCO compilation pipeline: from Modelica source to simulation executable, highlighting the intermediate representations, custom MLIR dialects, and integration with external solvers and runtime libraries.*

sections. Readers are encouraged to maintain a focus on the overarching structure and, where necessary, refer to the relevant chapters to gain a comprehensive understanding of the interconnections within the compilation pipeline.

4.3 Base Modelica Dialect

The `bmodelica` dialect is used to represent the Base Modelica source language and, therefore, the concepts that are fundamental to the DAE modeling process. The operations within this dialect are designed to capture the high-level semantics of the language, as well as to provide convenient representations for later stages of the compilation process.

System of DAEs The declarative nature of DAE-based languages is the most relevant aspect to be considered when defining an MLIR dialect for the compiler. The order in which equations are written does not affect the properties of the system nor changes the simulation behavior. Therefore, the intermediate representation chosen for a DAE system should not impose any order among the elements of this set.

The Base Modelica model class – representing the entire system of DAEs – is mapped on the `model` operation of the `bmodelica` dialect. Coherently with its source language counterpart, the MLIR operation is decorated with a name in the form of a symbol attribute. It also provides a single block for the declaration of variables and

equations, whose representation is discussed later in this section.

Variables Each variable belonging to a model is declared using the `variable` operation. Similarly to the `model` operation, it has a symbol attribute expressing the name of the variable. No Static Single Assignment (SSA) values are produced as a result of the declaration, with a symbol instead being used within the IR to reference the variable. The preference for symbols over SSA values has practical justifications. Indeed, MLIR operations may be characterized by the *isolated-from-above* trait, which forbids, inside their regions, to reference SSA values defined within parent scopes. Such a trait would make referencing variables harder in the case of SSA values declarations, and therefore possibly reduce interoperability with other MLIR dialects. The `variable` operation is also characterized by a `!bmodelica.variable` type describing the dimensionality, the nature, and the read-only traits (i.e., being a parameter or a constant) of the data. Dimensions are used to represent an array of elements, and can be omitted in the case of scalar variables. The `parameter` and `constant` specifiers are also optional. The nature of the individual elements is instead mandatory, and can be chosen from the already available MLIR built-in type system (e.g., `i64`, `f64`, etc.), or among the new types closely mapping the language syntax. This last category is composed of the `!bmodelica.bool`, `!bmodelica.int`, and `!bmodelica.real` types, which trivially correspond to the `Boolean`, `Integer`, and `Real` types of Base Modelica. In the examples provided within this and the following sections, we stick to the former category to keep the textual representation as compact as possible, but nonetheless can be replaced with their Base Modelica equivalents.

Equations The representation of equations must take several aspects into account, from effectively capturing high-level semantics to providing memory and computational efficiency throughout IR transformations. Multiple constructs are employed to achieve this multi-faceted goal, starting with the `equation_template` and `equation_instance` operations, which decouple the generic structure of an equation from the index space on which it is defined.

An `equation_template` operation has a single block containing the syntactic tree of the equation. The root of this syntactic tree is defined by the `equation_sides` operation, which takes two operands – each of them representing a list of expressions – that are in turn defined by two separate `equation_side` operations. As their names suggest, the `equation_sides` operation represents the concept of equality that is intrinsic to an equation, while the `equation_side` operation groups the values defining one side of the equation, given by one or possibly more expressions. If multiple operands are provided on the two equation sides, their quantity must be the same, and the equality implied by `equation_sides` is then assumed to hold among the elements at equal positions within the two lists.

Each expression is represented as an SSA use-def chain using SSA values defined only within the block of the equation template. The referencing to variables is performed through the `variable_get` operation, which enables the transition from the symbol semantics of variable declarations to the SSA semantics that characterize the equation body. This operation takes the name of the variable of interest and results in an SSA value containing its value. If the variable consists of a (possibly multidimensional)

array, then the result type is an MLIR tensor with the same dimensions and base type. In the case of a scalar variable, the result type corresponds instead to the data type of the variable. The time variable – which is always implicitly declared within a Base Modelica model – can be referenced using the `time` operation, which directly results in a scalar floating-point SSA value. Finally, among the SSA values that can be used for constructing the body of the equation, the `equation_template` operation provides a list of induction variables in the form of block arguments. If not empty, the SSA values of this list can be used to reference the generic indices of the array equation's structure being defined.

With the generic equation structure set in place, the actual declaration of an equation, or of an array of equations, is performed using an operation named `equation_instance`. This operation takes a reference to the identifier – expressed again in the form of SSA value – of an equation template, and is optionally enriched by an `indices` property providing the multidimensional range of indices of the equation.

While the `equation_template` operations are declared directly within the `model` operation, equation instances are placed within the unique block of an `initial` or of a `dynamic` operation. These last two constructs respectively express the validity of the contained equations for only the initialization of the system, or for the whole duration of the simulation.

The separation between equation structure and instantiation offers significant advantages in terms of memory efficiency. By defining the generic structure once, we avoid redundant representations of the mathematical operations that constitute the equation whenever additional properties are not equal across the iteration space. An example of such IR is shown in Listing 4.3, where the same equation template is instantiated three times with different index ranges that are also associated with different values of an additional attribute. Moreover, the benefits of this representation also extend to later compilation stages where equation bodies are converted into actual functions, as detailed in Section 5.2.6.

```
1  bmodelica.model @EquationInstanceExample {
2    %t0 = bmodelica.equation_template inductions = [%i] {
3      // Long equation body
4    }
5
6    bmodelica.dynamic {
7      bmodelica.equation_instance %t0 {indices = <[1,30]>, additional_property = "value1"}
8      bmodelica.equation_instance %t0 {indices = <[50,60]>, additional_property = "value2"}
9      bmodelica.equation_instance %t0 {indices = <[70,100]>, additional_property = "value3"}
10     // ...
11   }
12 }
```

Listing 4.3: *Example of repeated instantiation of the same equation structure. The separation of the equation body from its indices and additional properties allows to avoid code duplication for indices not sharing the same complete set of properties.*

Again on the topic of design decisions, the choice to adopt SSA semantics for the `equation_template` operation is motivated by the restricted contexts in which an `equation_instance` may appear (e.g., within the `initial` and `dynamic` operations). Unlike variables, which can be referenced from any operation within an equation body, the SSA value resulting from an `equation_template` operation is guaranteed, by design, to

be referenceable wherever required. This property makes SSA semantics preferable to the symbol-based approach, which would otherwise necessitate manual symbol table management and thus introduce potential sources of error.

An example of DAE system An example of `bmodelica` dialect can be seen in Listing 4.4, which represents the MLIR counterpart of the Base Modelica model of Listing 2.3. Each variable declaration within the Base Modelica source code has been translated into a `variable` MLIR operation accompanied by both its name and type. The `g` and `c` variables are both characterized by the `!bmodelica.variable<f64, parameter>` type. That is, each of them is a 64-bit scalar floating-point variable whose value – as indicated by the `parameter` keyword, is set only once during the initialization phase (i.e., when the initial values for the integration algorithm are computed). The variable named `T` is instead an array of 1000 64-bit floating-point elements, which are updated throughout the whole simulation. Each equation in Base Modelica is translated into a pair of `equation_template` and `equation_instance` MLIR operations, with the former being placed directly within the body of the model, and the latter inside an `initial` or a `dynamic` operation according to the initial / non-initial nature of the equation. For the sake of simplicity, in this thesis the assignments to parameters are treated as initial equations, as being performed only once during the initialization phase. In reality, additional dedicated operations are used to represent the value bindings performed inline with the variable declaration. The discussion of such details however transcends the scope of this work. Moreover, for brevity, only two equations have been fully reported in the MLIR code snippet, but the translation of the remaining ones operates along the same lines.

The variables `g`, `c`, and `T` are referenced within the equation bodies by using the `variable_get` operation. The variable `T`, being an array variable, is obtained in the form of a tensor having multiple elements. Its individual elements are then obtained using the `tensor_extract` operation, but could be seamlessly replaced with the `extract` operation of the core `tensor` dialect. The existence of the `tensor_extract` operation is justified by the additional need for slicing semantics – not used in this example – that are not provided by the core operation counterpart. The same applies to the `add`, `sub`, and `mul` operations, which could be replaced by their counterparts living in the `arith` dialect, which are sufficient for this example, but are unable to cover the whole semantics of Modelica. It should be noted that the numbering of elements within arrays is 1-based in both Modelica and Base Modelica, but 0-based in their MLIR counterpart. Therefore, accesses in the reported MLIR snippet are subject to an offset of `-1` that is automatically introduced during the translation process. The key concept of derivative – with respect to time – of a variable is mapped onto the `der` operation, which, in this example, is applied to a single element of variable `T`. The handling of this operation is discussed in more detail in Section 5.1.2.

Functions The Base Modelica language, as well as Modelica, also provides the ability to define functions. In the `bmodelica` dialect, this construct is represented through the `function` operation, which has a symbol attribute for the name of the function, a list of `variable` operations, and an `algorithm` operation containing the list of instructions to be executed. Unlike model variables, function variables can have unknown, runtime-

```
bmodelica.model @Wire {
  bmodelica.variable @g : !bmodelica.variable<f64, parameter>
  bmodelica.variable @c : !bmodelica.variable<f64, parameter>
  bmodelica.variable @T : !bmodelica.variable<1000xf64>

  // c * der(T[i]) = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i])
  %t0 = bmodelica.equation_template inductions = [%i] {
    %0 = bmodelica.constant 1 : index
    %1 = bmodelica.constant 2 : index
    %2 = bmodelica.variable_get @c : f64
    %3 = bmodelica.variable_get @g : f64
    %4 = bmodelica.variable_get @T : tensor<1000xf64>
    %5 = bmodelica.sub %i, %0 : (index, index) -> index
    %6 = bmodelica.tensor_extract %4[%5] : tensor<1000xf64>
    %7 = bmodelica.der %6 : (f64, f64) -> f64
    %8 = bmodelica.mul %2, %7 : (f64, f64) -> f64
    %9 = bmodelica.sub %i, %1 (index, index) -> index
    %10 = bmodelica.tensor_extract %4[%9] : tensor<1000xf64>
    %11 = bmodelica.sub %10, %6 : (f64, f64) -> f64
    %12 = bmodelica.mul %3, %11 : (f64, f64) -> f64
    %13 = bmodelica.tensor_extract %4[%i] : tensor<1000xf64>
    %14 = bmodelica.sub %13, %6 : (f64, f64) -> f64
    %15 = bmodelica.mul %3, %14 : (f64, f64) -> f64
    %16 = bmodelica.add %12, %15 : (f64, f64) -> f64
    %lhs = bmodelica.equation_side %8 : tuple<f64>
    %rhs = bmodelica.equation_side %16 : tuple<f64>
    bmodelica.equation_sides %lhs, %rhs : tuple<f64>, tuple<f64>
  }

  // c * der(T[1]) = g * (T[2] - T[1]) + 2 * g * (673.15 - T[1])
  %t1 = bmodelica.equation_template ...

  // c * der(T[1000]) =
  // g * (T[999] - T[1000]) + 2 * g * (293.15 - T[1000])
  %t2 = bmodelica.equation_template ...

  bmodelica.dynamic {
    bmodelica.equation_instance %t0 {indices = <[2,999]>}
    bmodelica.equation_instance %t1
    bmodelica.equation_instance %t2
  }

  // g = 0.00314785
  %t3 = bmodelica.equation_template inductions = [] {
    %0 = bmodelica.variable_get @g : tensor<f64>
    %1 = bmodelica.constant 0.00314785 : f64
    %lhs = bmodelica.equation_side %0 : tuple<f64>
    %rhs = bmodelica.equation_side %1 : tuple<f64>
    bmodelica.equation_sides %lhs, %rhs : tuple<f64>, tuple<f64>
  }

  // c = 0.2707936
  %t4 = bmodelica.equation_template ...

  // T[i] = 293.15
  %t5 = bmodelica.equation_template ...

  bmodelica.initial {
    bmodelica.equation_instance %t3
    bmodelica.equation_instance %t4
    bmodelica.equation_instance %t5 {indices = <[1,1000]>}
  }
}
```

Listing 4.4: *MLIR representation of the Base Modelica model of Listing 2.3 using the Base Modelica MLIR dialect.*

dependent, dimensions. The mutually exclusive `input` and `output` qualifiers indicate whether a variable value is received as input or its value is returned to the caller. Variables without any of these two qualifiers are considered – according to the Modelica specification – as *protected* variables, meaning that they are accessible only within the function body.

The semantics of the `algorithm` operation follow the ones that a programmer usually expects: the instructions are executed in the order they are written, `for` and `while` loops imply a runtime iteration, and the values of variables can be updated at any point of the function body. In the `bmodelica` dialect, the value of output and protected variables is set using the `variable_set` operation, which takes the name of the variable and the value to be set. Input variables can not be updated, as indicated by the Modelica specification.

An example of function in the `bmodelica` dialect can be seen in Listing 4.5. The

```
1  bmodelica.function @countMax {
2    bmodelica.variable @values : !bmodelica.variable<?xf64, input>
3    bmodelica.variable @max : !bmodelica.variable<f64, output>
4    bmodelica.variable @index : !bmodelica.variable<index, output>
5
6    bmodelica.algorithm {
7      %zero_f = bmodelica.constant 0.0 : f64
8      bmodelica.variable_set @max, %zero_f : (f64, f64) -> ()
9      %values = bmodelica.variable_get @values : tensor<?xf64>
10     %zero_idx = bmodelica.constant 0 : index
11     %dim = bmodelica.dim %values, %zero_idx : tensor<?xf64> -> index
12     %idx = bmodelica.alloca : <index>
13     bmodelica.store %idx[], %zero_idx : <index>
14
15     bmodelica.for condition {
16       %i = bmodelica.load %idx[] : <index>
17       %cond = bmodelica.lt %i, %dim : (index, index) -> i1
18       bmodelica.condition(%cond : i1)
19     } body {
20       %i = bmodelica.load %idx[] : <index>
21       %el = bmodelica.tensor_extract %values[%i] : tensor<?xf64>
22       %max = bmodelica.variable_get @max : f64
23       %cond = bmodelica.gt, %el, %max : (f64, f64) -> i1
24
25       bmodelica.if %cond {
26         bmodelica.variable_set @max, %el : f64
27         bmodelica.variable_set @index, %i : index
28       }
29
30       bmodelica.yield
31     } step {
32       %i = bmodelica.load %idx[] : <index>
33       %one = bmodelica.constant 1 : index
34       %inc = bmodelica.add %i, %one : (index, index) -> index
35       bmodelica.store %idx[], %inc : <index>
36       bmodelica.yield
37     }
38   }
39 }
```

Listing 4.5: *MLIR representation of a Base Modelica function that determines the maximum value of a one-dimensional array of floating-point numbers, and returns both the maximum value and the position of the first element with such a value.*

function, named `countMax`, determines the maximum value of a one-dimensional array of floating-point numbers, and returns both the maximum value and the position of the first element with such a value. Notice how the `for` operation does not entail the same declarative semantics of `for_equation`, but rather implies a runtime iteration. Its syntax has also been designed to accommodate the needs of imperative programming, allowing for more flexible control flow and state management by having three separated

regions to express the loop condition, body, and step.

4.4 Solver Dialects

The MLIR infrastructure naturally promotes modularity and high-level abstractions, making it particularly well-suited for leveraging existing numerical solvers without being constrained by their specific implementation details or Application Programming Interfaces (APIs). This architectural flexibility allows MARCO to integrate established and well-tested solvers while maintaining a clean separation between the high-level mathematical representation and the low-level solver interfaces. Currently, MARCO supports two solvers from the SUNDIALS software suite: IDA, which serves as a DAE solver, and KINSOL, which is used to compute numerical solutions for nonlinear algebraic equations and cyclic equation sets. In this section, we examine the MLIR dialects that have been developed to interface with them.

IDA Dialect The `ida` dialect serves as the bridge between the IDA solver and the MLIR infrastructure, enabling the seamless integration of the external tool into the IR. Through a dedicated set of operations, it is possible to instantiate an instance of the solver, configure its variables, and define the equations that it has to solve.

An instance of the IDA solver is declared via the `instance` operation, which establishes a symbolic reference for subsequent configuration steps. Differential and algebraic variables are respectively registered using the `add_differential_variable` and `add_algebraic_variable` operations. The association between differential variables and their derivatives is handled implicitly during this process.

Each variable declaration is accompanied by attributes specifying its dimensions (with scalars represented by empty dimensions) and by references to getter and setter functions. These functions, essential for runtime access to individual variable elements, are defined using the `sundials.getter` and `sundials.setter` operations. These operations reside in the separate `sundials` dialect, reflecting their shared utility across both the `ida` and `kinsol` dialects.

Listing 4.6 shows how these operations can be used with respect to the model of Listing 4.4. A property named `der` is attached to the model to map differential to derivative variables. In the context of this section, it can be interpreted as a way to implicitly declare the $der_T[i] = \dot{T}[i]$ relationship, valid for any $i \in [0, 999]$; its syntax and rationale are described in more detail in Section 5.1.2. The information about the differential variable `T` and its derivative `der_T` is provided using the `add_differential_variable` operation. Notice that `add_algebraic_variable` is not used, as the dynamic part of the model does not present any algebraic variable (`g` and `c` are indeed only computed during initialization, and are not recomputed during the numerical integration process).

Each equation solved by IDA is declared using the `add_equation` operation, optionally taking the iteration space of the equation itself. The actual behavior of the equation is then specified through a set of associated functions. Specifically, each equation must be accompanied by a residual function – computing the difference between the left- and right-hand sides of the equation – and a set of functions computing the partial derivative of the residual with respect to the state variables and algebraic variables. These functions are implemented as `residual_function` and `pder_function`

```
module {
  bmodelica.model @Wire der = [<@T, @der_T, {[0,999]}>] {
    bmodelica.variable @g : !bmodelica.variable<f64, parameter>
    bmodelica.variable @c : !bmodelica.variable<f64, parameter>
    bmodelica.variable @T : !bmodelica.variable<1000xf64>
    bmodelica.variable @der_T : !bmodelica.variable<1000xf64>
  }

  ida.instance @ida

  func.func @init() {
    ida.init @ida

    %T = ida.add_differential_variable @ida {
      dimensions = [1000],
      differentialGetter = @T_getter,
      differentialSetter = @T_setter,
      derivativeGetter = @der_T_getter,
      derivativeSetter = @der_T_setter
    }

    func.return
  }

  sundials.getter @T_getter(%i: index) -> f64 {
    %0 = bmodelica.variable_get @Wire::@T : tensor<1000xf64>
    %1 = tensor.extract %0[%i] : tensor<1000xf64>
    sundials.return %1 : f64
  }

  sundials.setter @T_setter(%v: f64, %i : index) {
    %0 = bmodelica.variable_get @Wire::@T : tensor<1000xf64>
    %1 = tensor.insert %v into %0[%i] : tensor<1000xf64>
    bmodelica.variable_set @T, %1 : tensor<1000xf64>
    sundials.return
  }

  sundials.getter @der_T_getter(%i: index) -> f64 {
    %0 = bmodelica.variable_get @Wire::@der_T : tensor<1000xf64>
    %1 = tensor.extract %0[%i] : tensor<1000xf64>
    sundials.return %1 : f64
  }

  sundials.setter @der_T_setter(%v: f64, %i : index) {
    %0 = bmodelica.variable_get @Wire::@der_T : tensor<1000xf64>
    %1 = tensor.insert %v into %0[%i] : tensor<1000xf64>
    bmodelica.variable_set @der_T, %1 : tensor<1000xf64>
    sundials.return
  }
}
```

Listing 4.6: *Example of variable declaration using the* `ida` *dialect. A differential variable and its derivative are simulatenously registered using the* `add_differential_variable` *operation. Both the variables are specified to be one-dimensional arrays with 1000 elements each. The accessor functions for the differential variable are named* `T_getter` *and* `T_setter`*, while the ones for the derivative variable are named* `der_T_getter` *and* `der_T_setter`*.*

operations, respectively, and are linked to their corresponding equations through the `set_residual_function` and `add_pder_function` operations.

In the example of Listing 4.7, the first equation is defined over the index range $[2, 999]$, while the second and third equations are scalar. The residual and partial derivative functions are defined accordingly, with the first equation's functions taking an index argument to account for the varying induction variable.

In the case of the partial derivative functions, an additional index argument is provided as an argument to identify which scalar element of `T` is to be considered as the independent variable. In MARCO, the partial derivatives are computed by applying the Automatic Differentiation (AD) technique [54, 29], whose implementation is detailed

```
module {
  bmodelica.model @Wire der = [<@T, @der_T, {[0,999]}>] {
    bmodelica.variable @g : !bmodelica.variable<f64, parameter>
    bmodelica.variable @c : !bmodelica.variable<f64, parameter>
    bmodelica.variable @T : !bmodelica.variable<1000xf64>
    bmodelica.variable @der_T : !bmodelica.variable<1000xf64>
  }

  ida.instance @ida

  func.func @init() {
    ida.init @ida
    %T = ...

    // c * der(T[i]) = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i])
    %eq0 = ida.add_equation @ida indices = {[2, 999]}

    ida.set_residual_function @ida, %eq0, @eq0_res
    ida.add_pder_function @ida, %eq0, %T, @eq0_pder_T

    // c * der(T[1]) = g * (T[2] - T[1]) + 2 * g * (673.15 - T[1])
    %eq1 = ida.add_equation @ida

    ida.set_residual_function @ida, %eq1, @eq1_res
    ida.add_pder_function @ida, %eq1, %T, @eq1_pder_T

    // c * der(T[1000]) = g * (T[999] - T[1000]) + 2 * g * (293.15 - T[1000])
    %eq2 = ida.add_equation @ida

    ida.set_residual_function @ida, %eq2, @eq2_res
    ida.add_pder_function @ida, %eq2, %T, @eq2_pder_T
  }

  // Residual function of eq0.
  ida.residual_function @eq0_res(%time: f64, %e_i: index, %v_i: index) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  // Partial derivative function of eq0 with respect to T.
  ida.pder_function @eq0_pder_T(%time: f64, %e_i: index, %v_i: index,
                                %alpha: f64) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  // Residual function of eq1.
  ida.residual_function @eq1_res(%time: f64, %v_i: index) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  // Partial derivative function of eq1 with respect to T.
  ida.pder_function @eq1_pder_T(%time: f64, %v_i: index, %alpha: f64) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  // Residual function of eq2.
  ida.residual_function @eq2_res(%time: f64, %v_i: index) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  // Partial derivative function of eq2 with respect to T.
  ida.pder_function @eq2_pder_T(%time: f64, %v_i: index, %alpha: f64) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }
}
```

Listing 4.7: *Example of equation declaration using the `ida` dialect. Three equations are declared with different index ranges: the first equation is declared having indices that range from 2 to 999, while the second and third equations are declared as scalar equations. Each of the three equations is associated with residual and partial derivative functions using the `set_residual_function` and `add_pder_function` operations. The functions `eq0_res` and `eq0_pder_T`, being associated to an array equation, receive the index `e_i` of the scalar equation being evaluated. All the partial derivative functions, instead, receive the index `v_i` of the scalar independent variable.*

in Section 5.1.1. The index of the independent variable is thus used to set to 1 the correct position within the array of seeds for variable `T`.

A final argument `alpha` is provided to partial derivative functions as part of the numerical integration method (see Section 5.2.4 for further details).

The KINSOL Dialect The `kinsol` dialect provides an interface between the MLIR infrastructure and the KINSOL solver, mirroring the design principles of the `ida` dialect. However, unlike IDA, KINSOL operates without a notion of time: it focuses on finding solutions to DAE systems at a fixed time point, employing a Newton-Krylov iterative method until convergence is achieved. For this reason, it omits constructs related to time derivatives. Variable declaration is performed using the only `add_variable` operation, which registers each algebraic variable handled by KINSOL, and the `pder_function` operation does not require the `alpha` parameter, which was instead present in the `ida` dialect.

```
module {
  runtime.model_name "Test"

  runtime.number_of_variables 4
  runtime.variable_names ["g", "c", "T", "der_T"]
  runtime.variable_dimensions [[], [], [1000], [1000]]
  runtime.variable_getters [@getter_g, @getter_c, @getter_T, @getter_der_T]

  runtime.variable_getter @getter_g() -> f64 { ... }
  runtime.variable_getter @getter_c() -> f64 { ... }
  runtime.variable_getter @getter_T(%i: i64) -> f64 { ... }
  runtime.variable_getter @getter_der_T(%i: i64) -> f64 { ... }
}
```

Listing 4.8: *Model metadata and variable getter declarations using the `runtime` dialect. The `model_name` operation declares the name of the model being "Test". The `number_of_variables` operation declares that the model has four (possibly array) variables, whose names are declared by the `variable_names` operation. The `variable_dimensions` operation specifies that `g` and `c` are scalar variables, while `T` and `der_T` are array variables with 1000 elements each. Finally, the `variable_getters` operation associates the functions used to fetch the array variable values, and each `variable_getter` implements such a function for a specific variable.*

```
module {
  runtime.scheduler @scheduler

  runtime.func @init {
    runtime.scheduler_create @scheduler
    runtime.scheduler_add_equation @scheduler, @eq, indices = {[0,999]}
  }

  runtime.equation_function @eq(%i: index) {
    // Implementation of the equation
  }

  runtime.func @updateAlgebraicVariables() {
    runtime.scheduler_run @scheduler
  }
}
```

Listing 4.9: *Scheduler configuration and equation execution using the `runtime` dialect. The `scheduler` operation declares a scheduler named `scheduler` whose implementation lives in the runtime libraries. The `scheduler_create` operation initializes the scheduler. The `scheduler_add_equation` operation adds an equation to the scheduler with indices from 0 to 999 included. Finally, the `scheduler_run` operation triggers the execution of all equations in the scheduler. Notice how the mentioned operations are used in different functions, which are expected to be called a different number of times during simulation execution (i.e., `init` is called once, while `updateAlgebraicVariables` is called at every time step).*

4.5 Runtime Dialect

The runtime environment – detailed later in Section 6 – serves two main purposes: it drives the execution of the simulation and provides essential data structures and algorithms required for efficient computation. The first purpose requires model-specific information, such as the number of variables and a way to access their values at runtime. The second purpose requires a mechanism for the compiled model to leverage the utilities provided by the runtime environment directly within the IR. A dedicated MLIR dialect, named `runtime`, has been designed to facilitate this interaction, abstracting the communication between the compiled model and the runtime system.

At the core of the `runtime` dialect is the `func` operation, whose semantics closely mirror those of the core `func` dialect: a symbol defines the function name, a region specifies the computations to be performed, and optional arguments and return types can be provided. This operation can be used in a two-fold manner. If the function is only declared (i.e., no body is provided in the IR), then the implementation is expected to be supplied by the runtime libraries. Otherwise, the function represents a callable entity that the runtime environment invokes during simulation to obtain the model-specific data.

More specialized operations are also defined to declare functions with a specific purpose, such as retrieving the value of a variable at runtime, but their rationale is more bound to the convenience of their representation rather than their real need. For example, as illustrated in Listing 4.8, the `model_name` operation specifies the model's name, `variable_getter` declares a function for retrieving a variable's value given its indices, and `variable_getters` lists all getter functions to be used at runtime.

Among the utilities provided by the runtime environment, particular importance is given to the *scheduler*, which is responsible for managing the multithreaded execution of equations. The usage of a scheduler and its configuration mirrors the approach of the `ida` and `kinsol` dialects. A scheduler is first declared in the IR using the `scheduler` operation, its equations are set using the `scheduler_add_equation` operation, and finally, the `scheduler_run` operation is used to trigger their processing. The sequential code implementing an equation's logic is defined within an `equation_function` operation, whose symbol name is given to the scheduler when the equation is added.

An example of these operations is provided in Listing 4.9, where a scheduler is configured with an equation of indices $[0, 999]$, and the processing is then requested within the `updateAlgebraicVariables` function.

CHAPTER 5

MLIR Compilation Pipeline

FOLLOWING the introduction of the MLIR dialects in Chapter 4, this chapter presents the principal transformations that constitute the compilation pipeline of MARCO. This pipeline translates a Base Modelica IR – produced after converting the AST into MLIR – into a low-level representation suitable for execution.

The compilation pipeline is organized into three main phases: canonicalization, model solving, and lowering. The *canonicalization* phase applies a sequence of transformations to standardize the IR, establishing well-defined assumptions that facilitate subsequent analyses and transformations. The *model solving* phase addresses the mathematical structure of the model, including causalization of the DAE system and algebraic manipulations. Finally, the *lowering* phase converts the IR into a form that maps directly to LLVM-IR, enabling compilation by the LLVM backend and integration with the simulation runtime environment.

It is important to note that the boundaries between these phases are not always clear-cut; several transformations have overlapping scopes and may influence the implementation of subsequent modifications. Moreover, a comprehensive and detailed description of the entire compilation pipeline is beyond the scope of this thesis. Many transformations, such as constant propagation and the elimination of redundant variables, are based on established compiler optimization techniques and, while beneficial for simulation performance, do not bring significant novelty in the context of Modelica compilation. The compilation pipeline is also subject to continuous evolution, with new transformations constantly being introduced and existing ones being refined. Even just on a graphical level, providing a complete and detailed description would result in an unwieldy representation whose readability would be compromised by the number of passes, their interactions, and which would nonetheless become quickly outdated after its inception. For these reasons, this chapter offers a general overview of the three

compilation stages and focuses only on a subset of transformations that either are particularly significant for the compiler as a whole, or that introduce novel techniques or representations. Readers interested in a comprehensive understanding of the continually evolving compilation pipeline are referred to the source code of the project.

5.1 Canonicalization

The initial MLIR representation is an almost one-to-one translation of the AST, and therefore still closely follows the original Base Modelica source code. This naive mapping implies that the IR has not yet been optimized, nor expressed in a form that makes further processing easier, more efficient, or even possible at all. Therefore, the first step in the frontend pipeline is to apply a set of canonicalization passes that transform the IR into a more suitable form.

5.1.1 Automatic Differentiation

Modelica – and possibly Base Modelica, depending on the future standardization of the language – supports the implicit definition of function derivatives, allowing users to declare that a function consists of the time derivative – also called *full* derivative – or partial derivative of another, without explicitly providing its implementation. This feature delegates the implementation of the derivative function to the compiler, which must analyze the base function and synthesize its derivative automatically.

To realize this capability, the compilation pipeline of MARCO employs the method of forward AD, a technique that systematically applies the chain rule to compute derivatives. Within the compiler, AD is not only essential for supporting implicit derivatives in the language, but also serves as a foundational technique for other components (e.g., the IDA and KINSOL solvers, and the planned index reduction for high-index DAE systems). For these reasons, a dedicated section is warranted.

The implementation of forward AD relies on the MLIR interface shown in Figure 5.1, which is implemented by all operations that may undergo differentiation. The

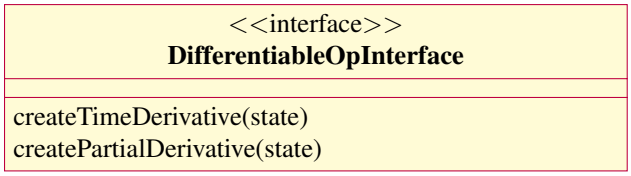

Figure 5.1: *UML representation of the* `DifferentiableOpInterface` *with its main methods.*

former interface defines two methods used to generate the time and partial derivatives of the operation. These methods receive and modify the *state* of the differentiation process. The state consists of a mapping between SSA values and their corresponding SSA derivatives, and of an equivalent mapping between more general operations that do not implement SSA semantics.

The differentiation of a function or an equation is performed by visiting the operations in the order they appear within the IR, and by calling the desired differentiation method. The operation-specific implementation can access the aforementioned mappings, and create the differentiated operation which is then memorized.

An example of differentiation of a function, with respect to the independent time variable and following the Modelica specification, is shown in Figure 5.2. The function `foo` computes the sum of two input variables, `x` and `y`, and stores the result in the output variable `z`. The function `der_foo` is the result of the differentiation of `foo` and computes both the value given by the function `foo` and its derivative. Only the latter is however the result of `der_foo`. The differentiation is performed by first copying the variables and declaring their derivatives, and then cloning each operation within the algorithm body, and differentiating it. The resulting chain of operations ending with `der_2` computes the time derivative of the sum of `x + y`, that is `x' + y'`. Variable `z` and its related computations – copied from the original algorithm – are no longer needed in the differentiated function and can be removed by later optimizations.

Of course, a preliminary initialization of the differentiation state may be required. When time-differentiating a function for the first time, the variables do not have a corresponding derivative, and therefore the state object tracking the process is initialized with an empty mapping. If the function is differentiated multiple times, however, the state must beforehand map the variables existing in the function to their corresponding derivatives: this is done by following the Modelica specification, which provides a set of rules to determine the name of the derivative variables (e.g., `der_x` for the first time derivative of `x`, and `der_i_x` for the i-th derivative of `x` when $i > 1$).

As previously discussed, the automatic partial differentiation of a function is also supported by the language. The implementation closely mirrors the one for time differentiation, with the key distinction that the newly introduced variables – referred to as *seeds* – are local to the differentiated function and are not provided by the caller. Each seed is initialized to either one or zero, depending on whether the corresponding variable is treated as independent or not. The remainder of the differentiation procedure follows the same principles as time differentiation.

Listing 5.1 illustrates partial differentiation of the function `foo` with respect to the variable `x`. In this example, the seed `der_x` is set to one, marking `x` as the independent variable, while `der_y` is set to zero. The resulting function `der_foo_x` computes both the original function value and its partial derivative with respect to `x`, returning only the latter as output.

5.1.2 Derivatives Materialization

The derivative operator – named `der` in Modelica and Base Modelica – is essential for modeling dynamic systems, where differential equations describe the evolution of the state of the system over time. However, this mathematical construct is not directly understandable by a machine and must be materialized into more practical, machine-friendly, representations. Indeed, when the simulation is executed, the value of a derivative must be stored somewhere in memory, so that later computations can fetch and use it while updating other variables. This memory location is reserved by declaring additional variables, one for each being the argument of the derivative operation. The derivative operations are then replaced with references to these newly inserted variables.

A specifically designed MLIR property is also attached to the model to inform later stages within the compiler about which variables have been inserted during this transformation. This property – named `der` – is represented as a list of tuples (a, b, i), in

```
bmodelica.function @foo {
  bmodelica.variable @x : !bmodelica.variable<f64, input>
  bmodelica.variable @y : !bmodelica.variable<f64, input>
  bmodelica.variable @z : !bmodelica.variable<f64, output>

  bmodelica.algorithm {
    %0 = bmodelica.variable_get @x : f64
    %1 = bmodelica.variable_get @y : f64
    %2 = arith.add %0, %1 : f64
    bmodelica.variable_set @z, %2 : f64
  }
}
```

```
bmodelica.function @der_foo {
  bmodelica.variable @x : !bmodelica.variable<f64, input>
  bmodelica.variable @y : !bmodelica.variable<f64, input>
  bmodelica.variable @der_x : !bmodelica.variable<f64, input>
  bmodelica.variable @der_y : !bmodelica.variable<f64, input>
  bmodelica.variable @z : !bmodelica.variable<f64>
  bmodelica.variable @der_z : !bmodelica.variable<f64, output>

  bmodelica.algorithm {
    %0 = bmodelica.variable_get @x : f64
    %der_0 = bmodelica.variable_get @der_x : f64
    %1 = bmodelica.variable_get @y : f64
    %der_1 = bmodelica.variable_get @der_y : f64
    %2 = arith.add %0, %1 : f64
    %der_2 = arith.add %der_0, %der_1 : f64
    bmodelica.variable_set @z, %2 : f64
    bmodelica.variable_set @der_z, %der_2 : f64
  }
}
```

Figure 5.2: *Example of automatic differentiation of a function computing the sum between two variables, with respect to the independent time variable. The function* `der_foo` *is the result of the differentiation of* `foo`.

```
1  bmodelica.function @der_foo_x {
2    bmodelica.variable @x : !bmodelica.variable<f64, input>
3    bmodelica.variable @y : !bmodelica.variable<f64, input>
4    bmodelica.variable @der_x : !bmodelica.variable<f64>
5    bmodelica.variable @der_y : !bmodelica.variable<f64>
6    bmodelica.variable @z : !bmodelica.variable<f64>
7    bmodelica.variable @der_z : !bmodelica.variable<f64, output>
8
9    bmodelica.algorithm {
10
11     %zero = bmodelica.constant 0.0 : f64
12     %one = bmodelica.constant 1.0 : f64
13     %der_x = bmodelica.variable_set @der_x, %one : f64
14     %der_y = bmodelica.variable_set @der_y, %zero : f64
15
16     %0 = bmodelica.variable_get @x : f64
17     %der_0 = bmodelica.variable_get @der_x : f64
18     %1 = bmodelica.variable_get @y : f64
19     %der_1 = bmodelica.variable_get @der_y : f64
20     %2 = arith.add %0, %1 : f64
21     %der_2 = arith.add %der_0, %der_1 : f64
22     bmodelica.variable_set @z, %2 : f64
23     bmodelica.variable_set @der_z, %der_2 : f64
24   }
25 }
```

Listing 5.1: *Example of automatic differentiation of a function computing the sum between two variables, with respect to the independent variable* `x`*. The function* `der_foo_x` *is the result of the partial differentiation of* `foo`*.*

which a and b are symbols and i is an optional set of indices. Such a tuple conveys the information of b being the new variable storing the derivative of variable a, for the indices given by i (if a is an array variable). The textual representation of this mapping between variables and derivatives is entirely decoupled from the actual implementation in memory. Indeed, the property does not practically consist of a list of tuples, but rather of multiple hashmaps putting each variable in relation with its derivative and differentiated indices. Such efficient representation can be obtained by directly operating on the operation, but does not need to be exposed in the textual version of the IR.

In Figure 5.3, we repropose part of the thermal model for a conductive wire – first introduced in Listing 4.4 – and illustrate how it is transformed by this IR transformation. A new variable named `der_T`, of the same type of `T`, has been declared, and the `der` property has been attached to the model operation to remember the relationship between the two variables. The `der` operation has been replaced with an access to the new `der_T` variable. More in detail, the same indices used to access the `T` variable are used to access the `der_T` variable. In case of array variables that are only partially differentiated, the new derivative variable would still be declared with the same dimensions as the differentiated variable, but only the relevant indices would be stored in the `der` property. Of course, additional dummy equations would be generated to ensure that the DAE system remains well-posed, but would not introduce any change in the model behavior.

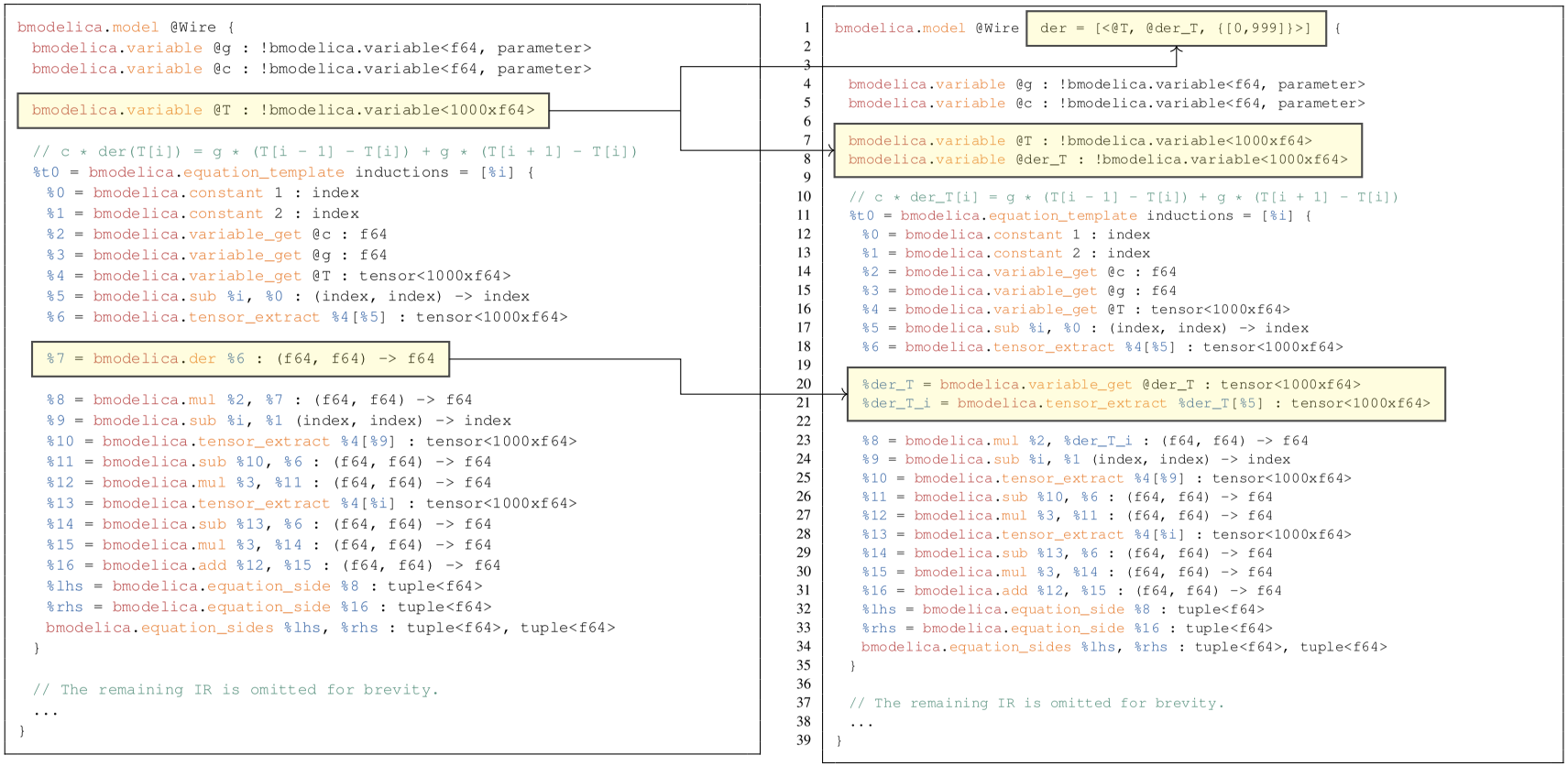

Figure 5.3: *Materialization of derivatives in the IR. On the left, the original operation uses the* `der` *operator to compute the derivative of* `T[i]`*. On the right, the transformation introduces a new variable* `der_T` *to store the derivative values, replaces the* `der` *operation with accesses to* `der_T`*, and attaches the* `der` *property to the model to record the mapping.*

5.2 Model Solving

Once the MLIR representation of the model has been transformed into its canonical form, the next phase involves solving the mathematical structure of the model.

Figure 5.4 provides an overview of the most relevant transformations that are implemented for this purpose. The pipeline consists of both mandatory and optional steps,

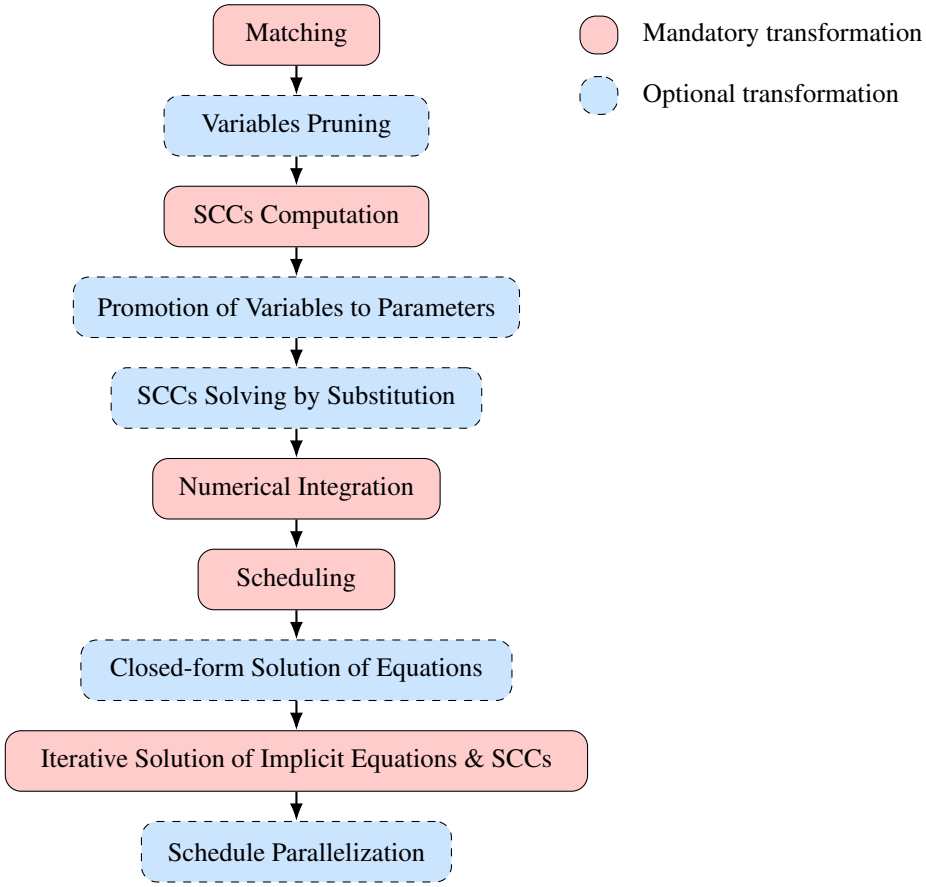

Figure 5.4: *Pipeline of transformation passes for the model solving phase, showing mandatory and optional transformation steps for mathematical processing and optimization.*

with optional transformations being performed based on the desired optimization level. Unlike the general mathematical framework discussed in Section 2.3, here the causalization steps – matching, SCC detection, and sorting – are always performed, independently of a DAE solver being applied or not. When a DAE solver is used, this approach allows the compiler to reduce the set of variables and equations that need to be handled externally, resulting in a more efficient simulation. This optimization is described in detail in the work by Scuttari et al. [63], and will be recalled in Section 5.2.4. Apart from this detail, the objectives of each transformation pass can be summarized as follows:

- **Matching**: Performs the matching of variables to DAEs, ensuring that each scalar variable is uniquely assigned to a scalar DAE.
- **Variables Pruning**: Removes the variables that are not used for the computation of the dimensions of interest, thus reducing the complexity of the subsequent steps.

- **SCCs Computation**: Identifies strongly connected components within the model, revealing cyclic dependencies among equations.
- **Promotion of Variables to Parameters**: Detects variables that depend solely on constants and parameters, and limits their computation to the initialization phase, thus avoiding redundant computations during numerical integration.
- **SCCs Solving by Substitution**: Applies symbolic substitution techniques to resolve cyclic equations within SCCs, aiming to derive closed-form solutions.
- **Numerical Integration**: Applies the chosen numerical integration technique to the system of DAEs. The applied algorithm may be completely internal to MARCO, or it may rely on external solvers.
- **Scheduling**: Orders the SCCs so that the data dependencies between the equations are respected during their sequential execution.
- **Closed-form Solution of Equations**: Attempts to analytically solve equations outside of cyclic relations that admit closed-form solutions, thereby reducing reliance on iterative numerical methods.
- **Iterative Solution of Implicit Equations & SCCs**: Employs iterative algorithms for equations and SCCs that cannot be solved analytically.
- **Schedule Parallelization**: Analyzes data dependencies within the sequential schedules to identify independent computations, enabling parallel execution.

5.2.1 Matching

This transformation pass is responsible for matching each scalar equation to a distinct scalar variable. It implements the array-aware matching algorithm described by Fioravanti et al. [26], which extends the Hopcroft-Karp algorithm by considering array variables and array equations as first-class citizens within the algorithm. As discussed in Section 3.3.1, the MCIM data structure has however been improved to adapt to a wider range of access patterns.

The matching algorithm is applied, in two separate phases, to both dynamic equations (i.e., the equations defining the behavior of the system) and initial equations (i.e., the equations defining the initial values of the variables). In the case of dynamic equations, the algorithm considers only the indices of variables for which a derivative does not exist. Differential variables are expected to be updated by the numerical integration algorithm, and therefore are not considered in the matching process. In the case of initial equations, instead, all the variables take part to the matching process as there is no distinction between differential and algebraic ones.

From an IR perspective, the results of the matching algorithm are attached to each array equation – represented by the `equation_instance` operation – as an attribute containing the symbol of the matched variable and its set of matched indices. For scalar variables, the set of matched indices is omitted.

As anticipated in Section 4.3, the separation between the equation body and the actual equation is crucial when solutions to the matching problem vary across the iteration space of the equation. If certain indices are matched to different variables, it suffices to

introduce multiple `equation_instance` operations, each with its own set of indices and matched variable information, without duplicating the equation body. This approach is not only memory-efficient, but also enables the caching of the list of accesses for the equation body, which remains consistent across all instances sharing the same structure.

Listing 5.2 shows a possible solution to the matching problem for our running example of Figure 5.3. It should be noted that the initial equation `der_T[i] = 0.0` was not

```
bmodelica.model @Wire {
  bmodelica.variable @g : !bmodelica.variable<f64, parameter>
  bmodelica.variable @c : !bmodelica.variable<f64, parameter>
  bmodelica.variable @T : !bmodelica.variable<1000xf64>
  bmodelica.variable @der_T : !bmodelica.variable<1000xf64>

  // c * der_T[i] = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i])
  %t0 = bmodelica.equation_template ...

  // c * der_T[1] = g * (T[2] - T[1]) + 2 * g * (673.15 - T[1])
  %t1 = bmodelica.equation_template ...

  // c * der_T[1000] = g * (T[999] - T[1000]) + 2 * g * (293.15 - T[1000])
  %t2 = bmodelica.equation_template ...

  bmodelica.dynamic {
    bmodelica.equation_instance %t0 {indices = <[2,999]>, match = <@der_T, {[2,999]}> }

    bmodelica.equation_instance %t1 { match = <@der_T, {1}> }

    bmodelica.equation_instance %t2 { match = <@der_T, {1000}> }
  }

  // g = 0.00314785
  %t3 = bmodelica.equation_template ...

  // c = 0.2707936
  %t4 = bmodelica.equation_template ...

  // T[i] = 293.15
  %t5 = bmodelica.equation_template ...

  // der_T[i] = 0.0
  %t6 = bmodelica.equation_template ...

  bmodelica.initial {
    bmodelica.equation_instance %t0 {indices = <[2,999]>, match = <@T, {[2,999]}> }

    bmodelica.equation_instance %t1 { match = <@T, {1}> }

    bmodelica.equation_instance %t2 { match = <@T, {1000}> }

    bmodelica.equation_instance %t3 { match = @g }

    bmodelica.equation_instance %t4 { match = @c }

    bmodelica.equation_instance %t5 {indices = <[1,1000]>, match = <@T, {[1,1000]}> }

    bmodelica.equation_instance %t6 {indices = <[1,1000]>, match = <@der_T, {[1,1000]}> }
  }
}
```

Listing 5.2: *Matched Wire model. Highlighted in yellow are the parts of the code that differ from the original code of Figure 5.3, and which convey the results of the matching process. Note that the matching is performed independently on the set of equations for the initial conditions and the set of equations for the dynamic behavior, and the results for the two sets are indeed different.*

present in the original code of Figure 5.3, but it is automatically generated by the canonicalization pipeline to obtain a DAE system that has a balanced number of variables and

equations.

5.2.2 Variables Pruning

The acausal, object-oriented nature of Modelica is advantageous because it allows modelers to connect components according to the requirements of the system being represented. This flexibility enables the construction of complex models by assembling reusable components, each of which may describe a wide range of physical properties and behaviors. However, the generic nature of these components means that their descriptions often include variables and equations that are not always relevant for every specific model instantiation.

When a model is instantiated, all variables and equations defined in the component descriptions are included in the resulting DAE system, regardless of whether they contribute to the properties of interest for the current analysis. This can lead to an unnecessarily large set of variables and equations, increasing the computational complexity. Based on these observations, Manzoni et al. [51] demonstrated the applicability of Tarjan's algorithm to identify the minimum set of equations that are necessary to compute the outputs of interest.

In the array-aware context of MARCO, the application of Tarjan's algorithm is performed on a dependency graph having as nodes the array variables of the DAE systems, similarly to the approach described in Section 3.3.2 for the detection of SCCs of equations (where, however, the nodes of the graph represented array equations). A node V_i in this graph is connected to another node V_j if any of the equations matched to V_i exhibit a dependency on variable V_j (i.e., if the equation needs to access V_j to compute V_i). As prescribed by the original algorithm, a dummy entry node is added to the graph, and an edge from this node to each variable of interest is created. This ensures that the algorithm will traverse the graph starting from the variables that are required for the simulation outputs, and will identify all other variables that are necessary to compute them. Finally, for each differentiated variable V_i, two edges are added to the graph: one from V_i to its derivative V_i', and one from V_i' back to V_i. These edges represent the implicit dependency between a variable and its derivative, and are obviously required by the numerical integration algorithm to work correctly.

The SCCs of the graph are then computed applying Tarjan's algorithm, starting from the dummy entry node. The variables that are part of the SCCs identified by the algorithm are retained, as they are necessary to compute the outputs of interest. The variables that are not part of any of these components are pruned from the model, as they do not contribute to the computation of the outputs, together with their matched equations.

5.2.3 SCCs Computation

This pass computes the SCCs within the DAE system by applying the multi-step array-aware algorithm proposed in Section 3.3.2. Similarly to the matching algorithm, the computation of SCCs is performed independently on both the sets of initial and dynamic equations. Each SCC is represented by a `scc` operation, which contains the list of equations – in the form of `equation_instance` operations – that are part of the SCC.

Before analyzing the IR resulting from the computation of SCCs on our running example, we first illustrate the usage of the `scc` operation through a simple system of

cyclic algebraic equations. Let us consider a model consisting of the algebraic equations

$$\begin{cases} a = b + c + 1 & (1) \\ b = a - c + 3 & (2) \\ c = 2 \cdot b & (3) \end{cases}$$

with Equation (1) being matched to the variable `a`, Equation (2) to `b`, and Equation (3) to `c`. As illustrated in Figure 5.5 the equations are cyclic and form a single SCC: Equation (1) depends on variable c, which is computed by Equation (3); Equation (3) depends on b, which is computed by Equation (2); Equation (2) depends on a, which is computed by Equation (1). The pass identifies this cycle and groups all three equations into a single `scc` operation, as shown in Listing 5.3.

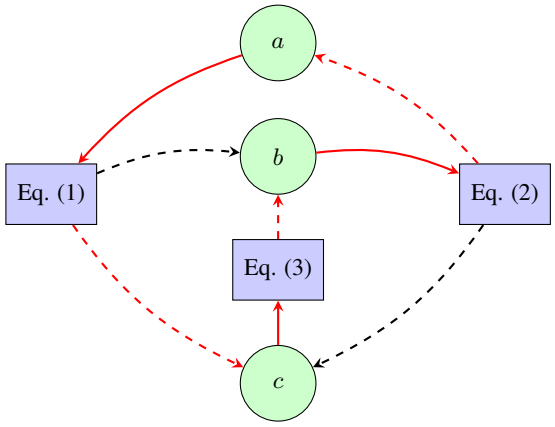

Figure 5.5: *Dependency graph for the system of equations (1) – (3). Equation nodes (blue rectangles) represent the three equations, and variable nodes (green circles) represent the variables a, b, and c. A solid arrow from a variable to an equation indicates that the equation is* matched *with that variable (i.e., the equation computes the value of that variable), while a dashed arrow from an equation to a variable indicates that the equation* reads *that variable (i.e., the value of the variable is needed by the equation). The red path shows the cycle formed by the three equations, and results in the SCC of Listing 5.3.*

The same algorithm can be applied to the DAE system of the Wire model of Listing 5.2 – on whose array equations the algorithm is able to efficiently reason – and results in the IR of Listing 5.4. In this case, no cycles are present within the system, and each of the original equations is simply wrapped by an independent `scc` operation.

If an array equation is only partially involved in a cycle, the algorithm is still able to identify the presence of such a cycle and to split the original array equation into multiple `equation_instance` operations, each one being part of only one `scc` operation. In this way, the equations that are involved in the cycle are grouped together in the same `scc`, while the equations that are not involved in any cycle are kept separate in different `scc` operations. A hypothetical example of this result is shown in Listing 5.5, where the equations with template `%t0` are involved in a cycle for indices $[51, 100]$, while they are not for indices $[1, 50]$.

```
bmodelica.model @CyclicAlgebraicSystem {
   bmodelica.variable @a : !bmodelica.variable<f64>
   bmodelica.variable @b : !bmodelica.variable<f64>
   bmodelica.variable @c : !bmodelica.variable<f64>

   // a = b + c + 1
   %t0 = bmodelica.equation_template ...

   // b = a - c + 3
   %t1 = bmodelica.equation_template ...

   // c = 2 * b
   %t2 = bmodelica.equation_template ...

   bmodelica.dynamic {
      bmodelica.scc {
         bmodelica.equation_instance %t0 {match = @a}
         bmodelica.equation_instance %t1 {match = @b}
         bmodelica.equation_instance %t2 {match = @c}
      }
   }

   // The remaining IR is omitted for brevity.
   // ...
}
```

Listing 5.3: *Result of SCC computation on a cyclic system of algebraic equations. All equations are grouped in a single* `scc` *operation, indicating the presence of a cycle among them.*

```
bmodelica.model @Wire der = [<@T, @der_T, {[0,999]}>] {
  bmodelica.variable @g : !bmodelica.variable<f64, parameter>
  bmodelica.variable @c : !bmodelica.variable<f64, parameter>
  bmodelica.variable @T : !bmodelica.variable<1000xf64>
  bmodelica.variable @der_T : !bmodelica.variable<1000xf64>

  // c * der_T[i] = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i])
  %t0 = bmodelica.equation_template ...

  // c * der_T[1] = g * (T[2] - T[1]) + 2 * g * (673.15 - T[1])
  %t1 = bmodelica.equation_template ...

  // c * der_T[1000] = g * (T[999] - T[1000]) + 2 * g * (293.15 - T[1000])
  %t2 = bmodelica.equation_template ...

  bmodelica.dynamic {
    bmodelica.scc {
      bmodelica.equation_instance %t0 {indices = <[2,999]>, match = <@der_T, {[2,999]}>}
    }

    bmodelica.scc {
      bmodelica.equation_instance %t1 {match = <@der_T, {1}>}
    }

    bmodelica.scc {
      bmodelica.equation_instance %t2 {match = <@der_T, {1000}>}
    }
  }

  // The remaining IR is omitted for brevity.
  // ...
}
```

Listing 5.4: *Result of SCC computation on the dynamic part of the Wire model of Listing 5.2. Each equation is wrapped in a separate* `scc` *operation, indicating that there are no cycles between them. Notice that, for brevity, only the SCCs of the dynamic part of the DAE system are shown.*

```
bmodelica.dynamic {
  bmodelica.scc {
    bmodelica.equation_instance %t0 {indices = <[51,100]>, match = <@x, {[51,100]}>}
    bmodelica.equation_instance %t1 {indices = <[151,200]>, match = <@y, {[151,200]}>}
  }

  bmodelica.scc {
    bmodelica.equation_instance %t0 {indices = <[1,50]>, match = <@x, {[1,50]}>}
  }
}
```

Listing 5.5: *Hyphotetical IR where only part of an array equation (with generic structure represented by `%t0`) contributes to a cycle. The remaining indices of the array equation are split into a separate equation instance and wrapped by a separate `scc` operation.*

5.2.4 Numerical Integration

We now proceed to highlight how the supported numerical integration algorithms are applied to the IR, analyzing the transformations and operations introduced by each numerical integration method within the compilation pipeline.

Although certain aspects of the interaction between the IR and the runtime environment may appear abstract at this stage, these details will be thoroughly clarified in Chapter 6, which is dedicated to the implementation of the runtime environment. The design of the compiler is inherently linked to the development of the runtime libraries, and as such, it is sometimes necessary to defer a comprehensive discussion of implementation specifics. This approach allows us to maintain a clear focus on the IR and its transformations, ensuring that each component is presented with appropriate context and precision when its discussion becomes relevant.

Explicit Euler

The application of the Explicit Euler method results in an IR whose example skeleton is shown in Listing 5.6. Given a model with a set of dynamic equations, already grouped according to the SCCs, the pass operates by first wrapping all the SCCs in a `schedule` operation, which is executed by the function named `updateNonStateVariables`. The execution of this function is performed by the runtime environment and, as such, is detailed in Section 6.2.2; once again, the design of a compiler is a process that is intertwined with the runtime execution of the compiled code, and therefore a linear explanation is sometimes difficult to achieve.

For each differential variable x, a new function-like operation is then instantiated to update its value using the $x_{i+1} = x_i + h \cdot x'_i$ formula. This operation – named `equation_function` – is parametric on the variable indices, which are provided by the callee in the form of an attribute that will later be materialized into actual function arguments. The invocation of this function is performed by an `equation_call` operation, which also conveys the independent nature of each call and therefore allows for potential parallel update of the scalar elements of x. To refer and update the variables within the model, two new operations – `qualified_variable_get` and `qualified_variable_set` – are introduced within the IR to perform read and write accesses to variables. These operations intentionally break the equational semantics of the language, but are nonetheless required to introduce, at some point within the pipeline, the imperative semantics of the simulation.

Also notice the usage of the opaque `schedule_block` operation, which encapsulates an arbitrary sequential logic to be executed as part of a schedule, and only permits reasoning about the overall set of accessed variables without actually exposing the way in which they are accessed.

The operations performing the update of state variables are inserted into a dedicated schedule that is executed by the `updateStateVariables` function, which is in turn called by the runtime environment. The time step h is passed as a parameter and, given the isolated nature of the schedule with respect to the function itself, the given time step value is stored in a global variable, so that it can be freely accessed at any point of the IR by other entities.

IDA

At the moment of writing, MARCO supports the usage of IDA as a DAE solver. The integration is performed by using the `ida` and `runtime` dialects, which were introduced in Section 4.4 and Section 4.5. On a high-level perspective, the modifications to the IR resemble the ones performed for Explicit Euler: the operations from the `runtime` dialect are used to declare the callbacks needed by the runtime environment, but their actual implementations will be interfacing with the IDA solver through the `ida` dialect. The AD infrastructure introduced in Section 5.1.1 is used to differentiate the equations of the DAE system, with the objective of computing the Jacobian matrix that is used by IDA to perform the numerical integration.

However, before analyzing the typical structure of the IR after this pass, we should recall how the results of the matching and SCCs analyses are used to reduce the number of equations and variables that are handled by the solver. This optimization is named *Clever DAE*, and consists of three sub-optimizations: the reduction of the DAE system, the *one-sweep* computation of the Jacobian matrix, and the pre-computation of always zero-valued partial derivatives. We now briefly recall these optimizations with the objective of understanding how the code generation is affected by them. The reader interested in a more mathematical background is referred to the work by Scuttari et al. [63].

The reduction of the DAE system operates by first identifying the set of equations that are either implicit, part of a non-trivial SCC, or matched to a derivative variable. This set contains the equations that are given to the external solver, while the remaining equations are handled internally. The optimization then proceeds to iteratively modify the set of externalized equations by removing the accesses to the variables that are computed by the equations of the second set, until no modification is possible. Each access to a variable is replaced by the explicit form of the equation that computes it, which is guaranteed to be obtainable by construction of the former sets of equations. From a mathematical point of view, this optimization brings the incidence matrix of the DAE into the form

$$IM = \begin{bmatrix} \mathbf{A} & \mathbf{0} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

where $\mathbf{A}$, $\mathbf{C}$, and $\mathbf{D}$ are matrices of appropriate dimensions, and the $\mathbf{0}$ is a zero-valued matrix. The incidence matrix is also said to be in Block Lower Triangular (BLT) form.

From the IR perspective, the equations given to the external solver are configured while initializing the dynamic solvers (i.e., within a `runtime.dynamic_model_begin` op-

```
module {
  memref.global @h : memref<f64>

  bmodelica.model @M der = [<@x, @der_x, <[0,999]>>] {
    bmodelica.variable @x : !bmodelica.variable<1000xf64>
    bmodelica.variable @der_x : !bmodelica.variable<1000xf64>

    // der_x[i] = sin(time)
    %t0 = bmodelica.equation_template ...

    bmodelica.dynamic {
      bmodelica.schedule @scheduleAlgebraic readVariables = [<@x, {[0,999]}>, <@der_x, {[0,999]}>,
          writtenVariables = [<@x, {[0,999]}>]] {
        bmodelica.scc {
          bmodelica.equation_instance %t0 {indices = <[0,999]>, match = <@der_x, {[0,999]}>}
        }
      }

      bmodelica.schedule @scheduleState {
        bmodelica.schedule_block {
          bmodelica.equation_call @euler_state_update_x {[0,999]} {independent = true}
        }
      }
    }
  }

  runtime.func @updateNonStateVariables() {
    bmodelica.run_schedule @scheduleAlgebraic
    runtime.return
  }

  // ~%x_i_new~ = x_i + h * der_x_i
  bmodelica.equation_function @euler_state_update_x(%i: index) {
    %x = bmodelica.qualified_variable_get @M::@x : tensor<1000xf64>
    %der_x = bmodelica.qualified_variable_get @M::@x : tensor<1000xf64>
    %h_memref = memref.get_global @h : memref<f64>
    %h = memref.load %h_memref[] : memref<f64>
    %x_i = tensor.extract %x, [%i] : tensor<1000xf64>
    %der_x_i = tensor.extract %der_x, [%i] : tensor<1000xf64>
    %0 = arith.mulf %der_x_i, %h : f64
    %x_i_new = arith.addf %x_i, %0 : f64
    bmodelica.qualified_variable_set @M::@x[%i], %x_i_new : tensor<1000xf64>
  }

  runtime.func @updateStateVariables(%h: f64) {
    %h_memref = memref.get_global @h : memref<f64>
    memref.store %h, %h_memref[] : memref<f64>
    bmodelica.run_schedule @scheduleState
    runtime.return
  }
}
```

Listing 5.6: *Skeleton of the IR for the Explicit Euler method. The* `scheduleAlgebraic` *and* `scheduleState` *schedule operations contain the equations and sequential logic needed to respectively update the algebraic and state variables. These schedules are invoked by the* `updateNonStateVariables` *and* `updateStateVariables` *functions, which are executed by the runtime environment.*

eration), and a callback function named `updateIDAVariables` is used to trigger, through the `ida.step` operation, their computation within the solver. Instead, the equations that are handled internally are grouped by a schedule operation that is then invoked by a callback function named `updateNonIDAVariables`.

The *one-sweep* computation of the Jacobian matrix is related to the approximation of the Jacobian matrix that is used by IDA, which reads as

$$\mathbf{J} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} + \alpha \frac{\partial \mathbf{F}}{\partial \dot{\mathbf{x}}}$$

with α being a value given by IDA at each time step. In the context of forward AD, each entry of the resulting matrix can be computed in one step rather than two, by setting to 1 the seed of the differentiated variable x, and to α the seed of the derivative variable $\dot{x}$. From the IR perspective, the seeds passed to the generic partial derivatives are just set accordingly by the caller.

Finally, the pre-computation of always zero-valued partial derivatives is performed by analyzing the set of variables used by each equation, and by informing the runtime environment about such accesses. During the execution of the simulation, the partial derivatives of an equation with respect to its non-accessed variables are automatically set to zero. From the IR perspective, the information about the accesses is given during the initialization of the IDA instance, through the `ida.add_variable_access` operation.

5.2.5 Scheduling

The scheduling pass is responsible for the ordering of the SCCs within the `schedule` operation, which we already briefly encountered in Section 5.2.4. The `schedule` operation is identified by a symbol, and its unique region wraps a sequence of operations that are executed as part of the simulation to update the variables of the DAE system. The execution of a schedule can be executed from any point of the IR by invoking the `run_schedule` operation, which takes as input the symbol of the schedule to be executed.

The `schedule` operation is designed to contain any arbitrary operation within its region, but the pass described in this section considers only the ones containing SCCs, with the purpose of ordering them as part of the causalization process. The ordering is performed by applying the algorithm described in Section 3.3.3. The direction of iteration for each dimension of an equation is represented through a property named `schedule`, which consists of a list of keywords that can be chosen among *any*, *forward*, and *backward*. In the first case, the indices for that dimension are allowed to be chosen in any order; in the second case, the induction variable must iterate starting from the lower bound and moving towards the upper bound of each iteration range; in the third case, the induction variable must iterate in the opposite direction, from the upper to the lower bound of each iteration range.

As an example, Listing 5.7 reports the result of the scheduling pass when applied to the `@scheduleAlgebraic` schedule of Listing 5.6.

5.2.6 Closed-form Solution of Equations

After the scheduling pass, the SCCs have been ordered so that the sequential processing of the equations would respect the data dependencies that exist between the variables. However, the actual processing of an equation is a process that is yet to be defined.

```
1 bmodelica.schedule @scheduleAlgebraic readVariables = [<@x, {[0,999]}>, <
      @der_x, {[0,999]}>, writtenVariables = [<@x, {[0,999]}>]] {
2   bmodelica.scc {
3     bmodelica.equation_instance %t0 indices = <[0,999]>,
4                 match = <@der_x, {[0,999]}>, schedule = [forward]
5   }
6 }
```

Listing 5.7: *Example result of the scheduling algorithm: during simulation, the equation's indices are to be visited in the forward direction (from* 0 *to* 999*).*

Indeed, their descriptive nature is still present, and an equation like $0 = 2x + y$ would not make any sense if interpreted as an assignment. This pass partially addresses this problem by best-effortly computing an analytical solution to the equations belonging to trivial SCCs (i.e., SCC containing a single equation). In the current implementation, the tentative solution is obtained by turning the two sides of the equation into a sum of products, and by then moving to the left-hand side of the equation only the matched variable. Following the previous example, the equation would be transformed into $x = -\frac{y}{2}$. Of course, this is not always possible, and the pass will simply leave the equation untouched if it cannot be solved in this way. If an explicit solution is obtained, a dedicated function-like operation named `equation_function` is used to implement the desired assignment parametrically with respect to the indices of the equation. This function-like operation is invoked by a `equation_call` operation taking as arguments the symbolic reference to the function and the iteration space of the equation. The call operation is wrapped in a `schedule_block` operation, as in Section 5.2.4.

An example of converted equations is shown in Listing 5.8. Three `schedule_block` operations have been declared to opaquely describe that a first assignment is performed to the variable x, a second assignment is performed to variable y, and a third assignment depending on x and y is performed to variable z. The actual assignments are encoded by the `equation_0`, `equation_1`, and `equation_2` functions, which access the variables of the original model both in read and write mode. Notice how the `schedule_block` operations are marked as parallelizable, meaning that subsequent optimizations are allowed to consider adjacent blocks as potentially groupable into a single parallel block, if the actual data-dependencies allow it. Such optimization is detailed in Section 5.2.8.

5.2.7 Iterative Solution of Implicit Equations and SCCs

As anticipated in the previous section, not all equations can be turned into an explicit assignment. Indeed, some of them are implicit (e.g., $x + sin(x) = 0$) or are involved in cyclic dependencies with other equations. In these cases, an approximate solution must be computed iteratively when the simulation is executed. The `kinsol` dialect is used to address this task by providing an interface to the KINSOL solver similarly to what is done with IDA (see Section 5.2.4).

Unlike IDA, however, the KINSOL solver has no concept of time, and therefore all differential and derivative variables are treated as algebraic ones. Each cycle is also solved independently, thus leading to a unique instance of KINSOL for each SCC.

From the point of view of the IR, the operations of the `kinsol` dialect are used to

```
module {
  bmodelica.model @M {
    bmodelica.schedule @schedule {
      bmodelica.dynamic {
        bmodelica.schedule_block writtenVariables = [@x], readVariables = [] {
          bmodelica.equation_call @equation_0
        } {parallelizable = true}
        bmodelica.schedule_block writtenVariables = [@y], readVariables = [] {
          bmodelica.equation_call @equation_1
        } {parallelizable = true}
        bmodelica.schedule_block writtenVariables = [@z], readVariables = [@x, @y] {
          bmodelica.equation_call @equation_2
        } {parallelizable = true}
      }
    }
  }

  bmodelica.equation_function @equation_0() -> () {
    // x := 0
    %0 = bmodelica.constant 0.0 : f64
    bmodelica.qualified_variable_set @M::@x, %0 : f64
  }

  bmodelica.equation_function @equation_1() -> () {
    // y := 1
    %0 = bmodelica.constant 1.0 : f64
    bmodelica.qualified_variable_set @M::@y, %0 : f64
  }

  bmodelica.equation_function @equation_2() -> () {
    // z := x + y
    %x = bmodelica.qualified_variable_get @M::@x : f64
    %y = bmodelica.qualified_variable_get @M::@y : f64
    %sum = bmodelica.add %x, %y : f64
    bmodelica.qualified_variable_set @M::@z, %sum : f64
  }
}
```

Listing 5.8: *Example of equations converted into explicit form: each schedule block is rewritten as a call to a generated equation function that encodes the solved assignment for the matched variable. The* `qualified_variable_set` *operations are used to reference the variables – and update their values – from outside the model they belong to.*

offload the equations to the KINSOL solver, mimicking the behavior of the `ida` dialect. The invocation of the external instances is performed within each `schedule_block` operation. An example is shown in Listing 5.9. Two instances of the KINSOL solver are declared, one for each SCC that remained unsolved. In the context of this example, let us suppose that the first SCC contained one implicit equation, and the second SCC contained two cyclic equations. For each SCC, a dedicated `schedule_block` operation is created to encapsulate the invocation of the respective KINSOL instance.

Similarly to IDA, each instance of the KINSOL solver is initialized and configured independently, and within a `dynamic_begin` operation that represents a sequence of code that is executed before starting the execution of numerical integration. Similarly, the instances are terminated within `dynamic_end` operations, which are called when the simulation has finished. If KINSOL is used during the initialization phase, the initialization and termination operations are instead placed within the similar `initial_begin` and `initial_end` operations, also belonging to the `runtime` dialect.

Notice how, differently from Listing 5.8, the two blocks are marked as non-parallelizable. This takes into account the already multithreaded implementation of the runtime environment when using KINSOL, and thus avoids the potentially uncontrolled spawning of threads during the simulation and the related cache-related inefficiencies.

```
module {
  kinsol.instance @kinsol_0
  kinsol.instance @kinsol_1

  bmodelica.model @M {
    bmodelica.schedule @schedule {
      bmodelica.dynamic {
        bmodelica.schedule_block writtenVariables = [@x], readVariables = [] {
          kinsol.solve @kinsol_0
        } {parallelizable = false}
        bmodelica.schedule_block writtenVariables = [@y, @z], readVariables = [@x] {
          kinsol.solve @kinsol_1
        } {parallelizable = false}
      }
    }
  }

  runtime.dynamic_begin {
    kinsol.init @kinsol_0
    %x = kinsol.add_variable ...
    %eq = kinsol.add_equation ...
    kinsol.set_residual ...
    kinsol.add_pder_function ...
    ...
  }

  runtime.dynamic_begin {
    kinsol.init @kinsol_1
    %y = kinsol.add_variable ...
    %z = kinsol.add_variable ...
    %eq0 = kinsol.add_equation ...
    %eq1 = kinsol.add_equation ...
    kinsol.set_residual ...
    kinsol.add_pder_function ...
  }

  runtime.dynamic_end {
    kinsol.free @kinsol_0
  }

  runtime.dynamic_end {
    kinsol.free @kinsol_1
  }
}
```

Listing 5.9: *Example of offloading unsolved SCCs to the KINSOL solver: two KINSOL instances are initialized by* `dynamic_begin`*, invoked by* `schedule_block`*, and destroyed by* `dynamic_end` *operations.*

5.2.8 Schedule Parallelization

After causalization, the data dependencies between SCCs become explicit, enabling the identification and scheduling of equations that can be executed in parallel. The approach adopted in this work is based on a modified version of Kahn's algorithm. In this algorithm, a directed graph is constructed so that each node represents a schedule block, and an edge from node i to node j indicates that block j depends on the output of block i. The algorithm proceeds by repeatedly identifying groups of nodes with no incoming edges. These blocks are independent and can be executed concurrently. After each group is scheduled, its nodes and outgoing edges are removed from the graph, and the process continues until all nodes have been processed.

At the IR level, groups of independent computations are represented using an operation named `parallel_schedule_blocks`. This operation contains a single region into which the opaque equations given by the original `schedule_block` operations are moved. Listing 5.10 provides an example of parallelized equations. The system of algebraic equations shown in (5.1) is scheduled into two partitions: equation 0 and equation

1 are grouped together, as they do not depend on each other and can be executed concurrently, while equation 2 is inserted into a separate partition that is executed after the first one.

$$\begin{cases} x = 0 & \text{(Equation 0)} \\ y = 1 & \text{(Equation 1)} \\ z = x + y & \text{(Equation 2)} \end{cases} \tag{5.1}$$

Although the example above is deliberately simple – and, in practice, the overhead of parallel execution would outweigh any performance gains for such a small system – it effectively illustrates the underlying concept. In practical applications, large systems of equations often contain many independent components, particularly in array-based models, where the individual scalar equations within an array may be independent from each other (e.g., as in the update of state variables in the explicit Euler method, as seen in Section 5.2.4).

```
1  bmodelica.schedule @schedule {
2    bmodelica.dynamic {
3
4      bmodelica.parallel_schedule_blocks {
5        bmodelica.schedule_block writtenVariables = [@x], readVariables = [] {
6          // Equation 0: x = 0
7          bmodelica.equation_call @equation_0
8        } {parallelizable = true}
9        bmodelica.schedule_block writtenVariables = [@y], readVariables = [] {
10         // Equation 1: y = 1
11         bmodelica.equation_call @equation_1
12       } {parallelizable = true}
13     }
14
15
16     bmodelica.parallel_schedule_blocks {
17       bmodelica.schedule_block writtenVariables = [@z], readVariables = [@x, @y] {
18         // Equation 2: z = x + y
19         bmodelica.equation_call @equation_2
20       } {parallelizable = true}
21     }
22
23   }
24 }
```

Listing 5.10: *MLIR representation of the system of equations 5.1 after the identification of equations that can be potentially executed in parallel. Equations 0 and 1 are identified as parallelizable and therefore grouped in the same* `parallel_schedule_blocks` *operation, while Equation 2 is kept separate as dependent on the variables matched by the other two.*

According to the underlying hardware architecture, the IR of Listing 5.10 can then be converted into the appropriate lower-level representations, possibly enabling parallel execution. This conversion is explained in Section 5.3.1.

5.3 Dialects Lowering

The last phase of the MLIR pipeline consists in the progressive conversion of the IR into LLVM-IR, which can then be further optimized and compiled to machine code by the backend of LLVM. From a high-level perspective, this phase is organized as in Figure 5.6. The `bmodelica` dialect is converted into the core MLIR dialects, which then undergo the bufferization process (i.e., tensors are replaced with memory buffers) and

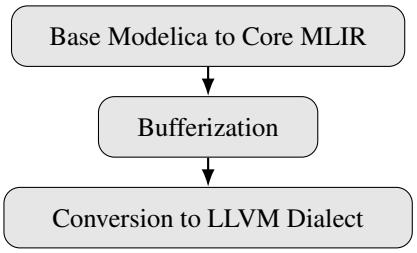

Figure 5.6: *Overview of the dialects lowering pipeline. The pipeline progressively lowers the high-level* `bmodelica` *dialect to core MLIR dialects, applies bufferization (converting tensors to memory buffers), and finally converts the result to the LLVM dialect. Optimizations like dead code elimination, loop fusion, and memory management improvements are applied throughout.*

are finally converted to the LLVM dialect, which represents a one-to-one mapping of LLVM-IR. Of course, the conversion of the dialects of MARCO to core MLIR dialects is not a trivial task, and many details should be taken into account. For example, the variables within a function may have known rank but unknown dimension sizes, thus allowing for runtime variations of the array sizes. Their conversion into MLIR dialects is delayed after bufferization, and therefore represents an exception to the aforementioned process. Moreover, several optimizations are again applied during the whole conversion. To name some, loops are fused, the deallocation operations are automatically inserted by the compiler, and heap allocations are turned into stack allocations when feasible.

Again, a detailed description of all the details goes beyond the scope of the work, and even though representing a considerable engineering effort, it does not provide a considerable source of novelty. An exception is the conversion of equation schedules, which is detailed here due to its key role in enabling the multithreaded execution of equations within the runtime environment.

5.3.1 Conversion of Equation Schedules

The conversion of equation schedules transforms the high-level schedule representation into executable code. This section describes two scheduling strategies: *sequential scheduling*, which is the base case used when parallelization is not performed, and *parallel scheduling*, which is employed when the detection of parallel equations is enabled, as explained in Section 5.2.8.

Both scheduling strategies rely on the conversion of equation functions, which transform the abstract equation representations into executable functions that operate over iteration ranges. An example of this conversion is given in Figure 5.7. The converted function takes the lower and upper bounds of the iteration space as parameters, and the inner loops iterate the original assignment over all the indices within the given bounds.

Sequential Scheduling

The *sequential scheduling* approach is used when the parallelization optimization is not performed, and thus the `parallel_schedule_blocks` operations are not present in the IR. In this case, the scheduled opaque blocks of computations are converted into calls to the

```
// Before conversion
bmodelica.equation_function @equation(%i: index) {
  // x[i] := 0
  %0 = bmodelica.constant 0.0 : f64
   bmodelica.qualified_variable_set @M::@x[%i], %0 : f64
}
```

```
// After conversion
func.func @equation(%lb: index, %ub: index) {
  %step = arith.constant 1 : index
  scf.for %i = %lb to %ub step %step {
    // x[i] := 0
    %0 = bmodelica.constant 0.0 : f64
    bmodelica.qualified_variable_set @M::@x[%i], %0 : f64
  }
}
```

Figure 5.7: *Example conversion of an equation function: the original* `bmodelica.equation_function` *is transformed into a* `func.func` *that accepts lower and upper iteration bounds and contains an* `scf.for` *loop iterating the original assignment over the index range.*

respective equation functions, or into the invocation of external solvers (e.g., KINSOL) whenever needed.

An example of sequential scheduling is given in Figure 5.8. The `schedule_block` operations, each containing an `equation_call` operation, are converted into calls to the respective equation functions, which take as input the lower (0) and upper (1000) bounds of the iteration spaces.

Parallel Scheduling

As explained in Section 5.2.8, when the parallelization optimization is enabled, the set of equations are partitioned into groups whose content can be executed concurrently at the array level and, according to access patterns, even at the scalar level.

The approach adopted in this work exploits parallelism through the runtime environment, which is responsible for managing the concurrent execution of equations within each group of independent equations. This strategy, denoted as *parallel scheduling*, delegates the actual task scheduling to the runtime environment, enabling it to adapt to the available computational resources.

The code generation process for parallel scheduling consists in generating the instructions needed to inform the runtime environment about the structure of the equations. Specifically, for each equation, the following information is provided: the associated equation function, the iteration boundaries of its multidimensional iteration space, and, optionally, whether the scalar equations within the array are independent from each other (i.e., whether the iteration space can be visited in any order). One runtime scheduler is instantiated for each group of independent equations (i.e., for each `parallel_schedule_blocks` operation), using the scheduler-related operations introduced in Section 4.5.

An example of parallel scheduling is given in Figure 5.9. Two runtime schedulers are configured and invoked. The configuration happens inside the `dynamic_begin` operations, whose content is executed only once at the beginning of numerical integration. Each scheduler is configured with the set of array equations that are independent from

```
// Before conversion
bmodelica.schedule @schedule {
  bmodelica.dynamic {
    bmodelica.schedule_block writtenVariables = [<@x, {[0,999]}>] readVariables = [] {
      bmodelica.equation_call @equation_0 {[0,999]}
    }
    bmodelica.schedule_block writtenVariables = [<@y, {[0,999]}>] readVariables = [] {
      bmodelica.equation_call @equation_1 {[0,999]}
    }
    bmodelica.schedule_block writtenVariables = [<@z, {[0,999]}>]
                             readVariables = [<@x, {[0,999]}>, <@y, {[0,999]}>] {
      bmodelica.equation_call @equation_2 {[0,999]}
    }
  }
}
```

```
// After conversion
bmodelica.schedule @schedule {
  bmodelica.dynamic {
    %lb = bmodelica.constant 0 : index
    %ub = bmodelica.constant 1000 : index

    func.call @equation_0(%lb, %ub)
    func.call @equation_1(%lb, %ub)
    func.call @equation_2(%lb, %ub)
  }
}
```

Figure 5.8: *Example of conversion with sequential scheduling: each* `schedule_block` *containing an* `equation_call` *is converted into a direct call to the corresponding equation function with concrete lower (0) and upper (1000) bounds, resulting in* `func.call` *invocations executed in order inside the* `dynamic` *operation.*

each other, together with their iteration ranges. The execution of each scheduler is then invoked at each time step by the `scheduler_run` operations. At the end of the simulation, the `dynamic_end` operations are executed, and the schedulers are destroyed by the `runtime.scheduler_destroy` operations.

It is worth noting that, in theory, it would be possible to statically map parallel computations onto OpenMP kernels or similar constructs during compilation. However, by delegating the scheduling to a custom pthread-based runtime environment, we allow for a more fine-grained control over the partitioning of equations. According to the actual computational heterogeneity of the equations, the user can choose to perform a more fine-grained partitioning: by decreasing the number of scheduled contiguous indices – and therefore by increasing the number of scheduled tasks – the user can increase the overall resource utilization up until a trade-off point where the function calls indirection overhead becomes predominant. Of course, identifying this critical point requires both knowledge of the model and experimental trials, but it nonetheless avoids the need to develop a cost model within the compiler, which would be difficult to implement and maintain in this first stage of compiler development. As a possible future improvement, this approach also enables the possibility of dynamically adapting the scheduling strategy (i.e., tuning the partitioning of iteration ranges) during simulation execution, based on the observed performance, instead of requiring manual intervention.

```
// Before conversion
bmodelica.schedule @schedule {
  bmodelica.dynamic {
    bmodelica.parallel_schedule_blocks {
      bmodelica.schedule_block writtenVariables = [<@x, {[0,999]}>] readVariables = [] {
            bmodelica.equation_call @equation_0 {[0,999]}
      }
      bmodelica.schedule_block writtenVariables = [<@y, {[0,999]}>] readVariables = [] {
            bmodelica.equation_call @equation_1 {[0,999]}
      }
    }
    bmodelica.parallel_schedule_blocks {
      bmodelica.schedule_block writtenVariables = [<@z, {[0,999]}>] readVariables = [<@x, {[0,999]}>, <@y,
            {[0,999]}>] {
        bmodelica.equation_call @equation_2 {[0,999]}
      }
    }
  }
}
```

```
// After conversion
module {
  runtime.scheduler @scheduler_0
  runtime.scheduler @scheduler_1

  bmodelica.model @M {
    bmodelica.schedule @schedule {
      bmodelica.dynamic {
        runtime.scheduler_run @scheduler_0
        runtime.scheduler_run @scheduler_1
      }
    }
  }

  bmodelica.dynamic_begin {
    runtime.scheduler_create @scheduler_0
    runtime.scheduler_add_equation @scheduler_0, @equation_0, indices = {[0,999]}
    runtime.scheduler_add_equation @scheduler_0, @equation_1, indices = {[0,999]}
  }

  bmodelica.dynamic_end {
    runtime.scheduler_destroy @scheduler_0
  }

  bmodelica.dynamic_begin {
    runtime.scheduler_create @scheduler_1
    runtime.scheduler_add_equation @scheduler_1, @equation_0, indices = {[0,999]}
  }

  bmodelica.dynamic_end {
    runtime.scheduler_destroy @scheduler_1
  }
}
```

Figure 5.9: *Example of conversion with parallel scheduling: independent equation groups are registered to runtime schedulers. Scheduler creation and equation registration occur inside* `dynamic_begin` *operations, schedulers are invoked at each time step via* `scheduler_run`*, and are destroyed in* `dynamic_end` *operations. Equation calls belonging to the same* `parallel_schedule_blocks` *operation result in the registration of the respective equation functions to the same scheduler, along with their iteration ranges.*

CHAPTER 6

Runtime Environment

THE runtime environment serves as the execution substrate for models compiled by the MARCO compiler. Its design is guided by two fundamental principles: providing a clean separation between model-specific code (generated by the compiler) and reusable infrastructure.

To achieve these goals, the runtime environment is structured around a layered architecture in which each library domain fulfills specific responsibilities while maintaining well-defined interfaces for interaction. In this chapter, we first present a high-level overview of the runtime environment's architecture and then delve into the specific libraries that comprise each domain.

6.1 Conceptual Overview

The runtime environment of MARCO consists of several libraries, whose design is closely linked to the structure of the compiler's IRs and the transformations applied during compilation. This section presents an overview of how these libraries interact to orchestrate simulation execution and support the code generated by the compiler.

The high-level interaction between the output of the compiler and the components of the runtime environment is depicted in Figure 6.1.

As shown in the figure, the compiler processes Modelica source code through various transformation stages (as detailed in Chapter 5), ultimately generating executable code that embodies the mathematical model. This compiled model, however, does not implement orchestration logic directly; instead, it focuses on providing model-specific and solver-specific information that the runtime environment uses to drive the simulation. More specifically, the compiled model exposes a well-defined set of functions that encapsulate specific computational tasks, such as evaluating equations and updat-

ing variables.

To execute a simulation, the runtime environment orchestrates the invocation of these functions and provides the necessary supporting infrastructure. This orchestration is performed through several libraries that can be conceptually categorized into four different domains, each addressing distinct aspects of simulation execution:

Driver Libraries manage the simulation lifecycle. They are responsible for the initialization and finalization of the simulation, as well as for the output of the simulation data and for the actual evolution of the system. Drivers implement the high-level control flow that allows the system to evolve over time, invoking the appropriate callbacks from the compiled model at the right moments. The actual callbacks and their semantics depend on the specific driver implementation, which may embody different simulation strategies (e.g., explicit Euler integration, IDA-based integration, etc.).

Utility Libraries provide foundational services that support simulation execution and management. These include facilities for command-line argument processing, profiling, data output (e.g., Comma-Separated Values (CSV) export), and parallel execution capabilities.

Integration Libraries serve as adapters to external numerical solvers, primarily those from the SUNDIALS suite (IDA for DAE systems and KINSOL for nonlinear equation solving). These libraries provide a consistent interface that shields the compiled model from the specifics of external solver APIs. When a driver delegates numerical integration to an external solver, the integration library establishes the necessary communication channels and ensures that solver callbacks are properly routed to the compiled model's functions.

Language Support Libraries implement language-level features required by the compiled model code, including Modelica built-in functions (e.g., mathematical operations, array manipulations, etc.) and memory management primitives. Unlike the other library domains that interact primarily with drivers, support libraries are invoked directly from the compiled model code whenever language-specific functionalities are needed.

6.2 Driver Libraries

This section examines the specific driver implementations that are currently available, beginning with the main driver that coordinates the overall simulation workflow and following with integration-algorithm-specific drivers.

6.2.1 Main Driver and Simulation Workflow

The main driver is an essential component that is always required during simulation execution, and operates in conjunction with a numerical-integration-specific driver. While numerical integration drivers handle the time-stepping logic for solving differential equations, the main driver orchestrates the complete simulation workflow from a higher level. Its responsibilities include initializing the runtime system, gathering structural information about the compiled model, requesting the solution of the algebraic model for

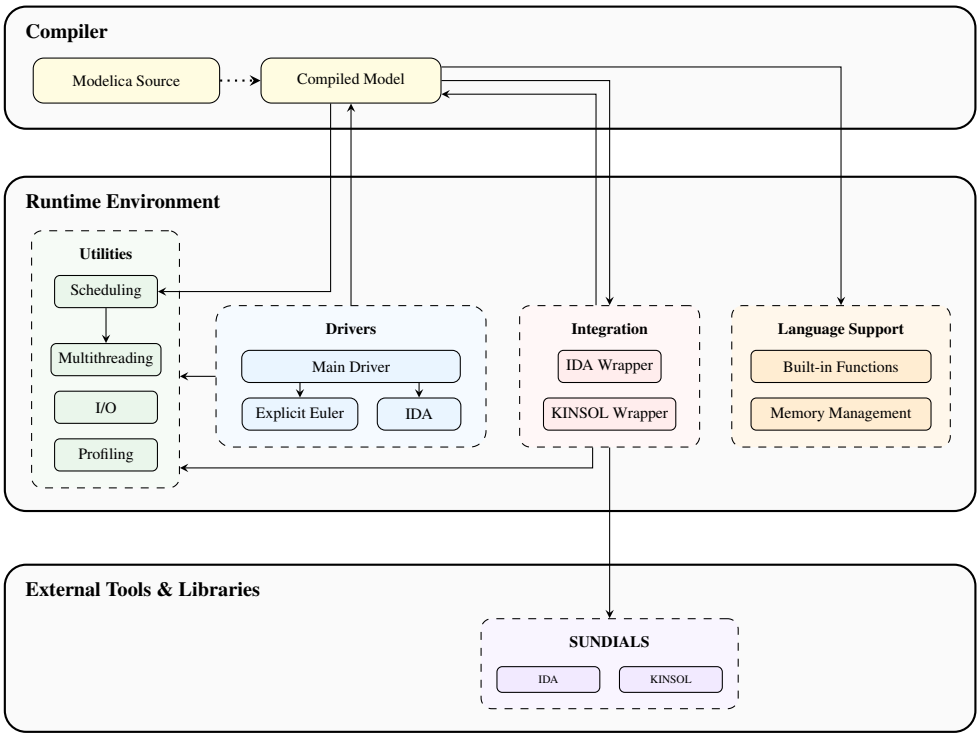

Figure 6.1: *The libraries composing the runtime environment together with their interactions between them and with both the compiled model and external libraries. A solid arrow between two components or groups of components indicates that the source component(s) leverage the feature provided by the target component(s). The dotted arrow from the Modelica source to the compiled model indicates that the latter is generated from the former through the compilation process.*

the initial conditions – which is independent from the chosen numerical integration algorithm – and delegating the evolution of the dynamic system to the selected numerical integration driver.

As illustrated in Figure 6.2, the main driver begins by gathering key structural properties of the compiled model, such as the total number of variables, their names, and the associations between differential variables and their derivatives. This information is obtained through functions automatically generated by the compiler, as summarized in Table 6.1. After initialization, the main driver instantiates a numerical integration

Function name	Description
`init`	Communicate the beginning of the simulation.
`setTime`	Set the current simulation time.
`icModelBegin`	Communicate the beginning of the computation of initial conditions.
`solveICModel`	Compute the initial conditions.
`icModelEnd`	Communicate the end of the computation of initial conditions.
`dynamicModelBegin`	Communicate the beginning of numerical integration.
`dynamicModelEnd`	Communicate the end of numerical integration.
`deinit`	Communicate the end of the simulation.

Table 6.1: *Functions used by the main driver of the runtime environment. Each function is expected to be implemented by the code generated by the compiler.*

driver and a data visualizer (i.e., the *printer* introduced in Section 6.3) based on user preferences. Simulation parameters, such as time step and stop time, are configured via command line arguments. Once configured, the main driver requests the computation of initial conditions and subsequently transfers control to the numerical integration driver, which handles the dynamic evolution of the system.

From an implementation perspective, the process involves additional details. For example, the selection of the numerical integration driver and printer is determined during compilation, rather than at the execution of simulation, with the appropriate libraries being chosen during the linking stage based on the chosen configuration. However, in this chapter the focus remains on the high-level design and behavior of the driver libraries, deliberately omitting low-level implementation details to maintain clarity and avoid unnecessary technical discussions.

6.2.2 Explicit Euler

The behavior of the explicit Euler driver is depicted in Figure 6.3, and relies on the callback functions described in Table 6.2. The driver operates by repeatedly updating

Function name	Description
`updateStateVariables`	Update the state variables according to the formula $x_{t+1} = x_t + \Delta t \cdot \dot{x}_t$.
`updateNonStateVariables`	Update the non-state variables according to the algebraic equations of the system.

Table 6.2: *Functions used by the explicit Euler driver of the runtime environment. Each function is expected to be implemented by the code generated by the compiler.*

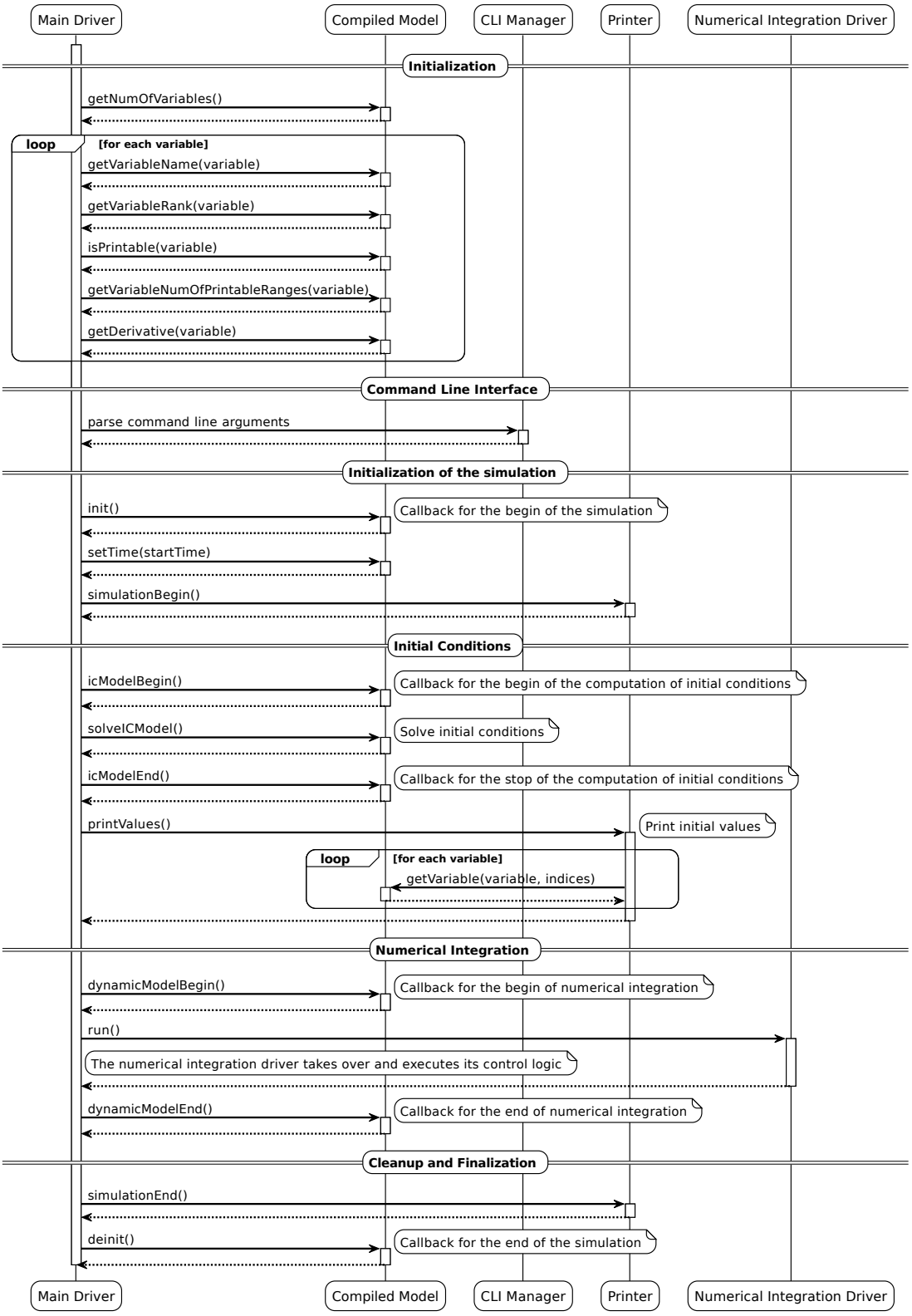

Figure 6.2: *Sequence of actions performed by the main driver of the runtime environment when the simulator is executed.*

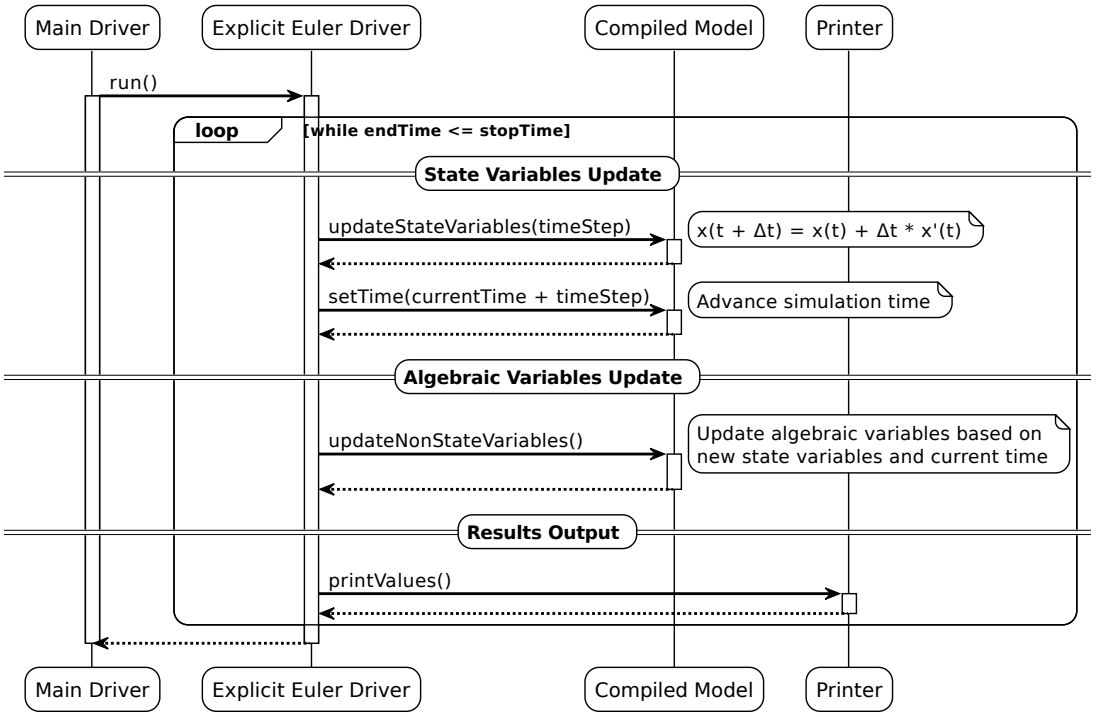

Figure 6.3: *Sequence of actions performed by the explicit Euler driver to compute the dynamic behavior of the compiled model.*

the state variables, advancing the time by a fixed step size, and updating the algebraic variables, until the simulation time has reached the requested stop time. At the end of each iteration, the values of the variables are communicated to the user.

The actual update of state and non-state variables is performed by invoking specific functions within the compiled model, namely `updateStateVariables` and `updateNonStateVariables`. Again, these functions are generated by the compiler, and their implementation may also involve additional interactions with other components of the runtime environment. For example, if the model is compiled with dynamic scheduling capabilities, the runtime schedulers will be invoked at each time step. This design allows the runtime driver to remain generic and agnostic to the specific details of the model, delegating all model-specific logic to the compiler-generated code.

6.2.3 IDA

The behavior of the IDA driver is depicted in Figure 6.4, and relies on the functions described in Table 6.3. The driver operates similarly to the explicit Euler one, with the

Function name	Description
`updateIDAVariables`	Update the variables handled by IDA.
`updateNonIDAVariables`	Update the variables handled by MARCO.
`getIDATime`	Get the time reached by IDA during the simulation.

Table 6.3: *Functions used by the IDA driver of the runtime environment. Each function is expected to be implemented by the code generated by the compiler.*

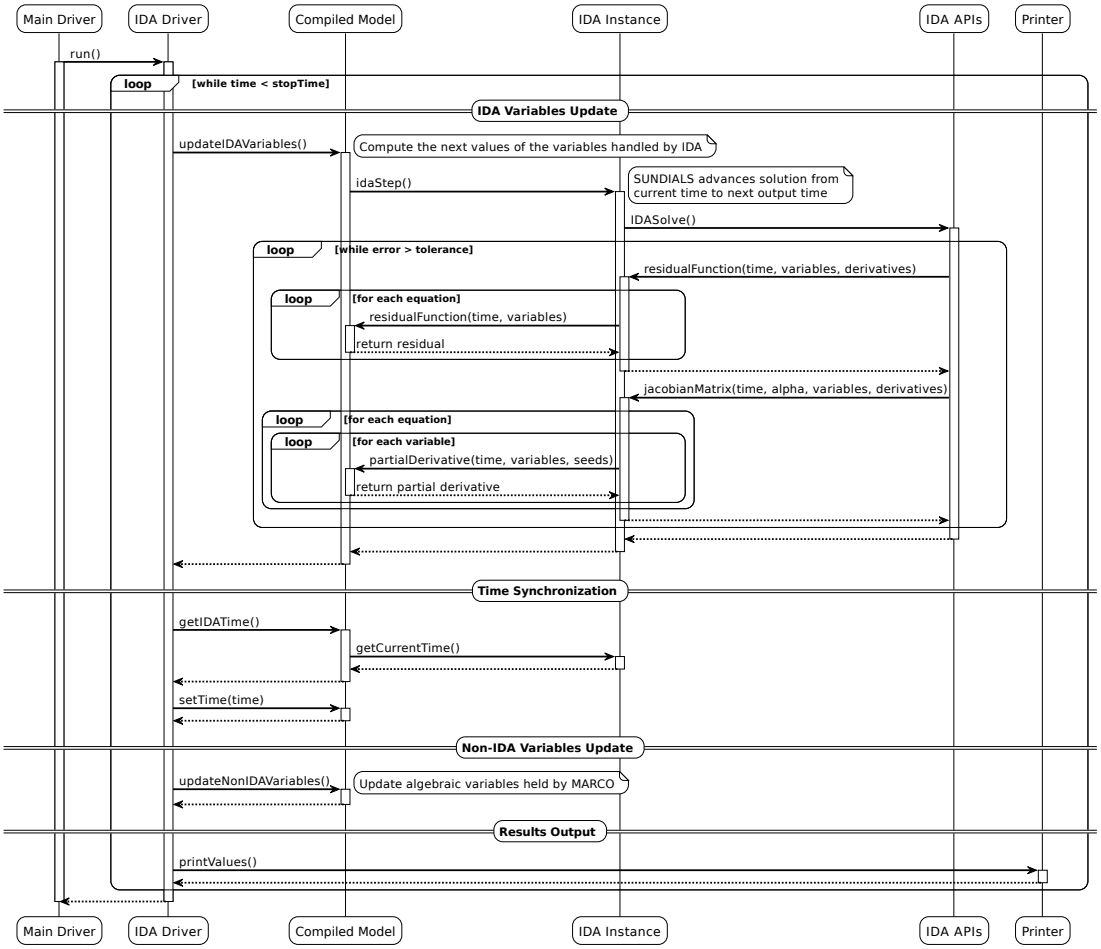

Figure 6.4: *Sequence of actions performed by the IDA driver to compute the dynamic behavior of the compiled model.*

main difference being that the time step is not fixed, but it is determined internally by the IDA solver based on the desired accuracy and actual simulation values. It should also be noted that the `updateIDAVariables` function updates not only state variables, but also derivative variables and, if the Clever DAE optimization [63] is disabled, algebraic ones.

The computations of residual values (i.e., the difference between the left-hand side and right-hand side of the equations) and of the Jacobian matrix are computed in a parallel manner, similarly to what is performed by the runtime schedulers described in Section 6.3. The computations – each independent from the others – are split into a number of groups obtained by multiplying the number of available threads by a tunable factor k. By adjusting the value of k, the load balancing can be improved, especially when the equations have heterogeneous computational costs.

6.3 Utility Libraries

The utility libraries form the layer that supports the execution and management of simulations, providing essential services such as profiling, command-line argument pro-

cessing, and multithreaded computation.

User interaction is handled by a centralized command-line argument manager. Each runtime component registers its own set of configurable options, enabling modular and flexible adjustment of simulation parameters. This design allows individual components to autonomously interpret and apply user-specified options. The profiling is built along the same lines: components can define custom profilers and register them with a centralized manager. At the conclusion of a simulation, this manager aggregates and presents the profiling data.

To facilitate simulation output, a versatile data visualization interface – termed the *printer* – is provided. This interface is designed for extensibility, allowing seamless integration of diverse output formats. The current implementation features a printer for the CSV format, enabling straightforward export and analysis of simulation results. However, the architecture is sufficiently flexible to support additional formats, such as graphical visualizations or network-based data streaming, thereby accommodating a wide range of post-processing and presentation needs.

From a performance perspective, the thread pool and scheduler are the most critical components. The thread pool enables concurrent execution of tasks. The scheduler orchestrates the distribution of equations across the available computing units.

Equations Scheduling A key innovation of this work is the introduction of multithreading capabilities, which distinguish MARCO from existing solutions that are limited to single-threaded simulations. As discussed in Section 5.2.8, MARCO automatically detects independent sets of equations and generates the necessary code to inform the runtime environment of these structural properties, utilizing the *Scheduler* class within the utility libraries.

The scheduler is configured via the runtime library APIs, receiving a collection of array equations, each assumed to be independent from the others. Additionally, each array equation may be annotated as scalarly independent, indicating that its iteration space can be traversed in any order.

Given a number of threads T, the scheduler partitions the equations into groups, each ideally containing up to $\lceil n/(T \cdot k) \rceil$ scalar equations, where n is the total number of scalar equations and k is a tunable factor for load balancing, accommodating variations in computational complexity. This threshold is treated as a soft limit: it may be exceeded when array equations lack scalar independence and their iteration spaces must be processed sequentially by a single thread.

The scheduling algorithm proceeds by iterating over the array equations. For equations marked as having independent indices, their iteration space is divided into contiguous ranges, each assigned to a group until the group's maximum capacity is reached. When a group is full, it is added to the schedule and a new group is started, continuing until all indices are scheduled. For equations without independent indices, the entire iteration space is assigned to a single thread, potentially exceeding the soft limit if necessary.

After constructing the groups, the scheduler evaluates whether multithreaded execution offers a tangible performance benefit. This is achieved by benchmarking a fixed number of numerical integration steps, which are first in a single-threaded manner and then using the computed multithreaded schedule. If the multithreaded approach yields

faster execution, it is adopted for the remainder of the simulation; otherwise, the scheduler reverts to single-threaded execution.

6.4 Integration Libraries

Integration libraries provide an additional abstraction layer over the APIs offered by external tools, facilitating seamless interaction within the runtime environment and the compiled model. In this work, two integration libraries have been developed to interface with the IDA and KINSOL solvers. Each solver is encapsulated within a dedicated C++ class, which can be instantiated and configured through its corresponding MLIR dialect (i.e., the `ida` and `kinsol` dialects, introduced in Section 4.4).

These classes expose specialized methods for configuration and execution, enabling the compiled model to interact with the solvers in a manner analogous to the scheduler. Although object-oriented design is not strictly necessary, it significantly enhances flexibility and maintainability, particularly when managing multiple solver instances, as is often required with KINSOL. For consistency and clarity, the same approach is adopted for the IDA solver.

The additional abstraction layer provided by the integration libraries also enables the extension of the underlying solvers with array-aware optimizations. Indeed, the computation of residuals and of the Jacobian matrix – essential operations for both solvers – is performed in a multithreaded fashion, with the employed partitioning strategy mirroring the logic described for the scheduler in Section 6.3. To ensure thread safety and uniqueness, the seeds used for computing partial derivatives are managed by a memory pool and generated in a manner that guarantees their distinctness across threads.

6.5 Language Support Libraries

Language support libraries provide implementations for functions that are not directly generated by the compiler. These libraries are particularly beneficial for operations where direct code generation in LLVM-IR would be impractical or detrimental to maintainability.

The functions offered by these libraries can be broadly classified into two categories: *built-in* functions, which are considered first-class functions by the Modelica language specification, and functions dedicated to memory management.

Since the initial development of the compiler in 2021, the dependency on the runtime environment for built-in functions has progressively diminished, with a growing preference for internal solutions that facilitate further optimization during compilation. Nonetheless, legacy implementations have been retained to provide fallback mechanisms in cases where internal approaches may prove inadequate.

In contrast, memory management functions remain a critical aspect of the runtime environment. They enable the customization of heap memory allocation and deallocation requested by the compiled model, typically by encapsulating standard routines such as `malloc` and `free` within custom functions. In previous works [62], this approach allowed evaluating the effect of garbage collection on simulation performance.

CHAPTER 7

Evaluation

THE evaluation of the proposed MLIR-based approach to the problem of compiling and simulating DAE models – and of the resulting MARCO compiler and its runtime environment – can be performed along two dimensions: the first one is the qualitative comparison between the proposed MLIR-based design and alternative approaches that employ a traditional, low-level IR, such as the one offered by the baseline LLVM framework; the second one is the quantitative comparison with existing Modelica compilers.

7.1 Qualitative evaluation of compiler design

The impact of the multi-level IR compared with the LLVM-IR lies primarily in the ease of development induced by the higher-level abstractions that MLIR can handle. In our use case, the need to represent, analyze, and transform equations – in addition to the procedural code – considerably raises the level of effort required to implement a compiler front-end entirely from scratch. The built-in tools of MLIR allow the software developer to focus on design issues while leaving support infrastructure to the framework. A low-level IR – such as LLVM-IR – would otherwise require reconstructing or analyzing these concepts from procedural code, a non-trivial and error-prone task. An example is the sorting of strongly connected components, which allows the algorithm proposed in Section 3.3.3 to operate on a single MLIR entity rather than refactoring the whole code of the model.

With respect to the DAE-based languages, the Modelica community has long recognized bottlenecks in large-scale model compilation with state-of-the-art solutions, including OpenModelica Compiler (OMC) [17]. Although several past and ongoing efforts [2, 58] have been directed toward this issue, such bottlenecks remain in OMC

to this day. By applying a multi-layer approach, MARCO facilitates the implementation of more efficient solutions to several critical problems and clearly demonstrates ground-breaking advantages. These gains are confirmed by the quantitative evaluation carried out in the following section.

7.2 Quantitative evaluation of compiler performance

The selection of Modelica compilers for benchmarking is subject to considerations on the source code availability and licensing terms. While commercial tools such as Modelon Impact and SimulationX are widely used, their closed-source nature prevents a transparent evaluation of their internal implementation and methodology. Furthermore, licensing terms for some tools (e.g., Dymola) explicitly prohibit performance benchmarking. As a result, this quantitative evaluation focuses on the comparison with OMC, whose implementation is open source and allows for a transparent and reproducible comparison with MARCO.

To thoroughly assess the performance of compilation and simulation, we consider three distinct models, each designed to highlight different aspects of the MARCO compiler and runtime environment. The first model is the wire example, which has been extensively discussed throughout this thesis. The second model extends the same physical principles to a three-dimensional silicon chip, enabling a more complex and realistic scenario. For this model, we propose three variants: one formulated as an ODE system, one as a DAE system, and one leveraging object-oriented features. This diversity allows us to evaluate the impact of different modeling paradigms and abstractions on both compiler and simulation performance. The third model focuses on thermal exchanges in a set of heat exchangers using methanol as the working fluid. This model is relevant for testing the correct execution and efficiency of user-defined functions, as it requires the computation of fluid properties during simulation. By including this model, we ensure that the benchmarks cover not only equation-based modeling but also the challenges associated with function evaluation, which are common in the modeling of cyber-physical systems. All models have been simulated using both the explicit Euler method and the IDA solver to provide a comprehensive comparison across the currently supported integration algorithms. The benchmarks are designed to systematically evaluate both the compiler and the generated simulator (i.e., the compiled model together with the runtime environment) as the number of equations in the model increases. Each model is implemented in a parametric fashion, allowing the user to quickly change the desired discretization granularity or model structure, and evaluate how compilation and simulation times scale with respect to the number of equations.

Hardware and Software Configuration All benchmarks were conducted on a dedicated CPU server partition of the IDUN HPC cluster [67]. For each experiment, 32 cores of an Intel Xeon Gold 6348 processor (2.60 GHz) and 128 GB of RAM were allocated. The cluster nodes run Rocky Linux 9.4 as the operating system; however, all benchmarks were executed within a containerized environment based on Debian 12 to ensure reproducibility and isolation.

All software components were built from source using recent rolling releases. Specif-

ically, LLVM was built from commit *ce6d227*[1], MARCO from commit *3ed4f34*, the MARCO runtime libraries from commit *67799c4*, and OMC from commit *e499b20*. The benchmark suite[2] and the Docker image of the compiler [3] used for this evaluation are also publicly available.

Wire Model This model corresponds to the one presented in Listing 4.4. As anticipated in Section 2.6, this model implements a physical description of thermal conduction along a copper wire. The wire is modeled as a one-dimensional object, partitioned in a configurable number of segments, each of which is described by a single temperature variable. The model was previously used in the work of Agosta et al. [6] to assess the scaling capabilities of OMC in the case of array-oriented models, and therefore represents a good starting point for the evaluation of both MARCO and the progress made by OMC since then.

The number of segments in the wire, and therefore the number of equations in the model, is controlled through the parameter `N`. Table 7.1 reports the tested values of `N` together with the corresponding number of equations in the model and time measurements. The results are also graphically represented in Figure 7.1.

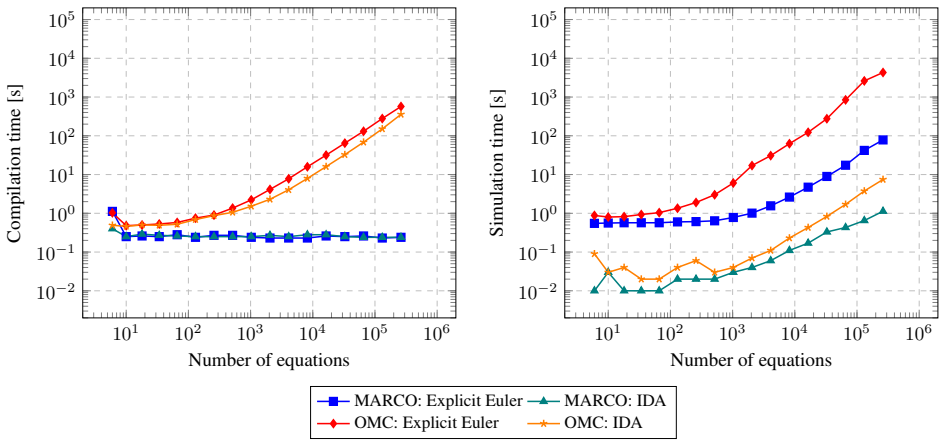

Figure 7.1: *Benchmark results for the wire model: log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

The results in Table 7.1 and Figure 7.1 highlight a marked difference in compilation and simulation performance between MARCO and OMC. MARCO consistently achieves a constant compilation time of approximately 250 ms, independent of the model size. This outcome is consistent with the proposed algorithms, which allow the compiler to process the array constructs of the model without the need for unrolling. In contrast, OMC displays a linear growth in compilation time as the number of equations increases, reaching up to 570 s for the largest benchmarked model (approximately 262 k equations) with the explicit Euler integration method. Although direct comparison with earlier studies is complicated by differences in hardware and the absence of

[1] The MARCO front-end integration, along with minor constraint relaxations within MLIR, requires a fork of the LLVM repository, available at `https://github.com/marco-compiler/llvm-project`. This commit is aligned with upstream LLVM commit *4a8bb08*.

[2] `https://github.com/mscuttari/polimi-phd-thesis-benchmarks`

[3] ghcr.io/marco-compiler/marco-prod-debian-12:3ed4f34

		MARCO			
		Explicit Euler		IDA	
N	Equations	Compilation	Simulation	Compilation	Simulation
4	6	1.12 s	0.55 s	0.40 s	0.01 s
8	10	0.25 s	0.56 s	0.25 s	0.03 s
16	18	0.26 s	0.57 s	0.29 s	0.01 s
32	34	0.25 s	0.57 s	0.27 s	0.01 s
64	66	0.28 s	0.57 s	0.26 s	0.01 s
128	130	0.24 s	0.60 s	0.25 s	0.02 s
256	258	0.27 s	0.61 s	0.25 s	0.02 s
512	514	0.27 s	0.64 s	0.25 s	0.02 s
1024	1026	0.24 s	0.78 s	0.25 s	0.03 s
2048	2050	0.23 s	1.01 s	0.27 s	0.04 s
4096	4098	0.23 s	1.56 s	0.25 s	0.06 s
8192	8194	0.23 s	2.62 s	0.28 s	0.11 s
16 384	16 386	0.26 s	4.72 s	0.28 s	0.17 s
32 768	32 770	0.25 s	8.93 s	0.25 s	0.33 s
65 536	65 538	0.26 s	17.4 s	0.24 s	0.43 s
131 072	131 074	0.23 s	41.97 s	0.24 s	0.65 s
262 144	262 146	0.24 s	78.28 s	0.24 s	1.13 s

		OMC			
		Explicit Euler		IDA	
N	Equations	Compilation	Simulation	Compilation	Simulation
4	6	1.03 s	0.88 s	0.49 s	0.09 s
8	10	0.48 s	0.80 s	0.46 s	0.03 s
16	18	0.50 s	0.82 s	0.50 s	0.04 s
32	34	0.53 s	0.93 s	0.49 s	0.02 s
66	66	0.58 s	1.04 s	0.52 s	0.02 s
128	130	0.75 s	1.35 s	0.69 s	0.04 s
256	258	0.90 s	1.90 s	0.87 s	0.06 s
512	514	1.36 s	3.01 s	1.06 s	0.03 s
1024	1026	2.22 s	6.05 s	1.52 s	0.04 s
2048	2050	4.13 s	17.06 s	2.30 s	0.07 s
4096	4098	7.78 s	30.74 s	4.05 s	0.11 s
8192	8194	15.86 s	62.61 s	7.94 s	0.23 s
16 384	16 386	31.70 s	122.98 s	16.03 s	0.43 s
32 768	32 770	64.61 s	275.60 s	32.63 s	0.83 s
65 536	65 538	130.93 s	845.70 s	68.43 s	1.70 s
131 072	131 074	278.32 s	2610.62 s	150.76 s	3.76 s
262 144	262 146	570.25 s	4299.98 s	357.17 s	7.46 s

Table 7.1: *Benchmark results for the wire model: compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *µs. The stop time is set to* 0.4 *s for both explicit Euler and IDA.*

detailed historical records, the observed trend in OMC aligns with the findings reported by Agosta et al. [6]. A similar pattern emerges when employing the IDA solver, with OMC requiring more than 357 s to compile the largest model.

Simulation times further underscore the efficiency of MARCO. With the explicit Euler method, MARCO exhibits linear scaling with respect to the number of equations, with a simulation time of approximately 78 s for the largest configuration, whereas OMC requires 4299 s under the same conditions. The adoption of the IDA solver reduces simulation times for both software solutions due to its adaptive time step, yet MARCO maintains a substantial advantage, achieving simulation times that are up to seven times faster than those of OMC in the largest tested scenario.

Silicon Chip Model Expanding upon the foundational wire model, which encapsulates the essential principles of one-dimensional thermal conduction, the evaluation proceeds to a more intricate and realistic scenario: the thermal modeling of a three-dimensional silicon chip. The need for this model stems from the increasing power densities observed in modern microprocessors, for which precise simulation of thermal dynamics is mandatory for maintaining reliability and performance. Excessive heat dissipation, if not properly managed, can indeed lead to component degradation or inefficiencies.

State-of-the-art approaches address this challenge by using co-simulation frameworks [71], wherein the thermal behavior of the chip and its cooling infrastructure are computed by distinct simulation tools. Integrating both the chip and its cooling solution within a unified Modelica model, however, would lead to a more cohesive and streamlined description, avoiding the complexities and overhead associated with co-simulation. To this end, several benchmark models have been developed in Modelica.

The first set of benchmarks represents a three-dimensional silicon chip discretized along all spatial axes. Similarly to the segments of the wire, each volume is modeled as a lumped thermal capacitance connected to six thermal conductances (one for each spatial direction). The top surface of the chip interfaces with a two-dimensional fixed temperature source, emulating a fixed-temperature heatsink, while half of the bottom surface is connected to a constant power source representing the active semiconductor area.

Three distinct implementation paradigms are considered: an ODE system, a DAE system, and an object-oriented formulation. The textual representations of these models are provided in Appendix A under the names `ThermalChipODE`, `ThermalChipDAE`, and `ThermalChipOO`. The ODE and object-oriented variants were also previously utilized by Agosta et al. [6] in the initial development of the HiPerMod benchmark suite.

The ODE and DAE representations are harder to implement with respect to the object-oriented version, as they require the explicit formulation of all equations and careful handling of boundary conditions and connectivity. The object-oriented variant greatly simplifies the implementation, as the model can be described using just three for loops, one for each spatial direction. This approach allows for flexible connectivity, enabling heat sources or other thermal objects of arbitrary shape to be attached to specific portions of the chip's outer faces, such as representing active semiconductor areas. Unconnected outer faces are automatically treated as thermally insulated, thanks to the default connection equations (`Q_flow = 0`) generated by the compiler for unconnected thermal ports. Nevertheless, this ease of representation comes at the cost of generating

a considerably higher number of equations with respect to the ODE and DAE cases, as each face-to-face connection between adjacent blocks results in a small system of linear equations corresponding to the series connection of the two half-conductances in that direction.

Performance results for the three `ThermalChip` model variants are presented in Table 7.2, Table 7.3, and Table 7.4, with corresponding visualizations provided in Figure 7.2, Figure 7.3, and Figure 7.4. The scaling behavior of the compilation time

				MARCO			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	88	1.89 s	1.36 s	1.20 s	0.03 s
6	6	4	188	1.06 s	1.48 s	1.10 s	0.03 s
8	8	4	328	1.01 s	1.51 s	1.08 s	0.04 s
12	12	4	728	1.04 s	1.69 s	1.08 s	0.06 s
15	15	5	1358	1.05 s	2.00 s	1.08 s	0.15 s
18	18	6	2276	1.03 s	2.34 s	1.05 s	0.33 s
24	24	8	5192	1.03 s	3.31 s	1.08 s	3.08 s
33	33	11	13 076	1.03 s	6.59 s	1.06 s	20.57 s
39	39	13	21 302	1.04 s	9.56 s	1.09 s	75.47 s
54	54	18	55 412	1.03 s	21.31 s	1.08 s	1022.32 s
66	66	22	100 196	1.00 s	40.86 s	1.06 s	4664.24 s
84	84	28	204 632	1.03 s	96.05 s	1.07 s	$>$ 3 h
114	114	38	506 852	1.03 s	143.53 s	1.93 s	$>$ 3 h
144	144	48	1 016 072	1.03 s	175.26 s	1.05 s	$>$ 3 h
183	183	61	2 076 326	1.01 s	258.25 s	1.10 s	$>$ 3 h

				OMC			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	88	1.15 s	1.60 s	0.67 s	0.09 s
6	6	4	188	0.90 s	1.80 s	0.88 s	0.02 s
8	8	4	328	1.16 s	2.21 s	1.24 s	0.04 s
12	12	4	728	1.92 s	3.04 s	2.13 s	0.10 s
15	15	5	1358	3.02 s	4.67 s	3.69 s	0.21 s
18	18	6	2276	4.93 s	8.64 s	6.17 s	0.53 s
24	24	8	5192	12.38 s	28.77 s	14.90 s	3.52 s
33	33	11	13 076	35.05 s	67.27 s	41.27 s	32.87 s
39	39	13	21 302	60.74 s	105.24 s	72.37 s	98.06 s
54	54	18	55 412	204.8 s	328.59 s	235.86 s	1426.51 s
66	66	22	100 196	477.91 s	1082.49 s	519.76 s	4669.04 s
84	84	28	204 632	1819.99 s	2800.56 s	1908.69 s	$>$ 3 h
114	114	38	506 852	$>$ 1 h	-	$>$ 1 h	-

Table 7.2: *Benchmark results for the silicon chip model in ODE formulation (`ThermalChipODE`): compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *µs. The stop time is set to* 0.4 *s for both explicit Euler and IDA.*

closely mirrors the one observed in the wire model: MARCO consistently achieves compilation in constant time, while OMC displays a linear increase as the model size grows. For the explicit Euler method, simulation time scales linearly with the number of equations for both tools; however, MARCO maintains a substantial performance advantage across all tested configurations. Notably, the activation of multithreaded execution in MARCO becomes apparent at approximately 10^5 and 10^6 equations, according to the model variant. Simulations employing the IDA solver yield nearly equivalent

				MARCO			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	328	1.63 s	1.48 s	1.10 s	0.03 s
6	6	4	704	0.69 s	1.57 s	1.05 s	0.03 s
8	8	4	1224	0.68 s	1.67 s	1.06 s	0.04 s
12	12	4	2696	0.69 s	1.96 s	1.11 s	0.09 s
15	15	5	5108	0.69 s	2.26 s	1.05 s	0.15 s
18	18	6	8648	0.72 s	2.95 s	1.09 s	0.41 s
24	24	8	19 976	0.72 s	4.89 s	1.05 s	2.84 s
33	33	11	50 828	0.76 s	10.84 s	1.06 s	21.82 s
39	39	13	83 156	0.70 s	16.09 s	1.04 s	95.11 s
54	54	18	217 736	0.71 s	49.49 s	1.08 s	1126.13 s
66	66	22	394 952	0.70 s	179.56 s	1.08 s	4444.98 s
84	84	28	809 096	0.70 s	261.48 s	1.08 s	> 3 h
114	114	38	2 010 056	0.70 s	272.43 s	3.56 s	> 3 h
144	144	48	4 036 616	0.69 s	353.99 s	1.08 s	> 3 h
183	183	61	8 260 628	0.69 s	643.52 s	1.07 s	> 3 h

				OMC			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	328	1.36 s	1.94 s	0.75 s	0.06 s
6	6	4	704	1.29 s	2.69 s	1.00 s	0.05 s
8	8	4	1224	1.82 s	3.81 s	1.43 s	0.04 s
12	12	4	2696	3.62 s	9.80 s	2.71 s	0.12 s
15	15	5	5108	6.75 s	22.76 s	5.19 s	0.25 s
18	18	6	8648	12.01 s	37.49 s	8.95 s	0.64 s
24	24	8	19 976	28.52 s	77.77 s	21.06 s	3.59 s
33	33	11	50 828	76.02 s	206.26 s	55.69 s	23.89 s
39	39	13	83 156	127.36 s	338.62 s	95.25 s	99.55 s
54	54	18	217 736	364.95 s	1697.75 s	272.91 s	852.49 s
66	66	22	394 952	727.45 s	3347.83 s	595.19 s	5108.44 s
84	84	28	809 096	CLI Error	-	CLI Error	-

Table 7.3: *Benchmark results for the silicon chip model in DAE formulation (`ThermalChipDAE`): compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *µs. The stop time is set to* 0.4 *s for both explicit Euler and IDA. The "CLI Error" indications in the results of OMC are due to the large amount of command-line arguments (i.e., names of the generated C files) passed by OMC to the C compiler.*

				MARCO			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	1040	2.45 s	1.74 s	2.38 s	0.03 s
6	6	4	2330	1.68 s	2.01 s	2.29 s	0.05 s
8	8	4	4136	1.68 s	2.21 s	2.24 s	0.04 s
12	12	4	9296	1.69 s	3.31 s	2.26 s	0.10 s
15	15	5	17 873	1.67 s	4.75 s	2.26 s	0.16 s
18	18	6	30 626	1.71 s	6.48 s	2.35 s	0.41 s
24	24	8	71 720	1.68 s	11.82 s	2.25 s	2.87 s
33	33	11	184 544	1.69 s	31.00 s	2.28 s	17.81 s
39	39	13	303 389	1.69 s	50.49 s	2.28 s	73.15 s
54	54	18	800 450	1.69 s	158.49 s	2.31 s	953.92 s
66	66	22	1 457 090	1.68 s	469.66 s	2.27 s	4290.09 s
84	84	28	2 995 280	1.67 s	422.72 s	2.26 s	$>$ 3 h
114	114	38	7 466 210	1.68 s	497.65 s	3.14 s	$>$ 3 h
144	144	48	15 023 240	1.73 s	983.38 s	2.32 s	$>$ 3 h
183	183	61	30 792 869	1.67 s	2400.6 s	2.29 s	$>$ 3 h

				OMC			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	1040	3.92 s	24.34 s	1.93 s	0.13 s
6	6	4	2330	12.06 s	62.32 s	4.12 s	0.14 s
8	8	4	4136	13.13 s	116.44 s	5.97 s	0.31 s
12	12	4	9296	30.67 s	261.13 s	16.61 s	0.63 s
15	15	5	17 873	88.89 s	546.45 s	34.12 s	1.64 s
18	18	6	30 626	149.00 s	888.69 s	63.41 s	3.59 s
24	24	8	71 720	354.77 s	2374.61 s	204.36 s	16.19 s
33	33	11	184 544	925.45 s	8083.67 s	646.11 s	144.12 s
39	39	13	303 389	1862.61 s	$>$ 3 h	$>$ 1 h	-
54	54	18	800 450	$>$ 1 h	-	$>$ 1 h	-

Table 7.4: *Benchmark results for the silicon chip model in object-oriented formulation (`ThermalChipOO`): compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *µs. The stop time is set to* 0.4 *s for both explicit Euler and IDA.*

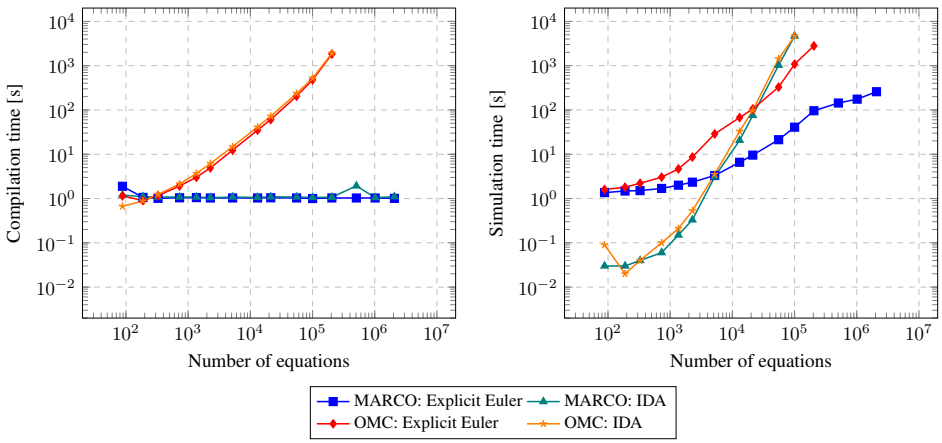

Figure 7.2: *Benchmark results for the silicon chip model in ODE formulation (`ThermalChipODE`): log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

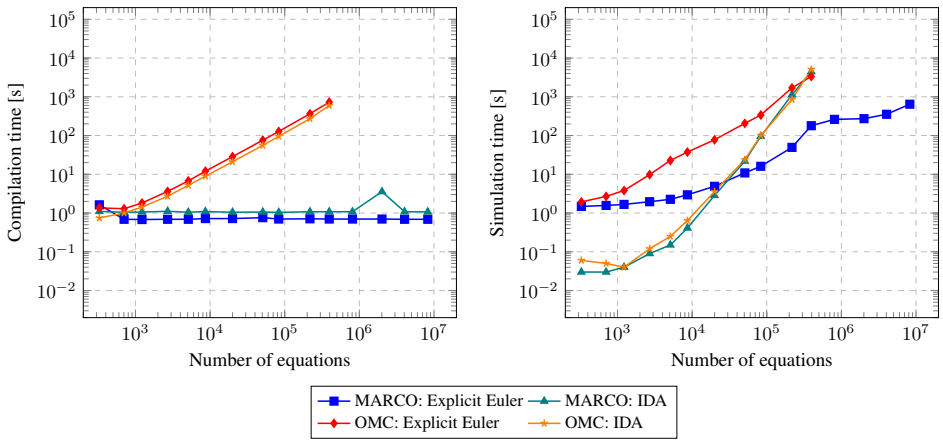

Figure 7.3: *Benchmark results for the silicon chip model in DAE formulation (`ThermalChipDAE`): log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

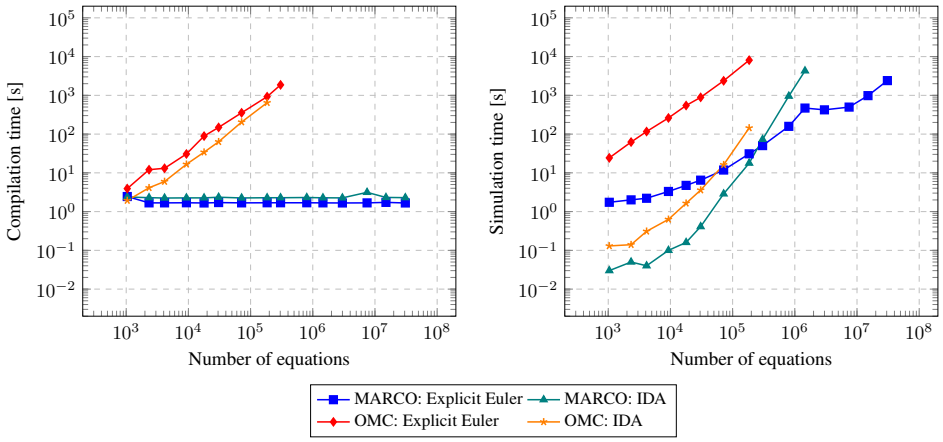

Figure 7.4: *Benchmark results for the silicon chip model in the object-oriented formulation (`ThermalChipOO`): log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

performance for both tools in the ODE and DAE variants, while being one order of magnitude in favor of MARCO in the largest configurations of the object-oriented formulation. This discrepancy is due to the Clever DAE optimization, which leads to a reduction of the externalized system that is more significant with respect to the other two variants. It should also be noted that compilation with OMC fails for DAE-variant models exceeding 809 096 equations, due to the excessive number of command-line arguments passed to the C compiler for the translation of the generated C files. This limitation, imposed by the operating system, highlights the scalability challenges inherent in the flattening approach adopted by OMC.

The object-oriented variant of the silicon chip is further extended to incorporate a more realistic cooling system. This is modeled as a series of one-dimensional channels applied to the top surface of the chip. The enhanced model, denoted as `ThermalChipCoolingOO`, introduces additional complexity, as the non-linear equations describing the convective heat transfer within the cooling channels require the use of the KINSOL solver to iteratively compute their solution.

The time measurements for the `ThermalChipCoolingOO` model are reported in Table 7.5 and visualized in Figure 7.5. The compilation times are similar to the basic

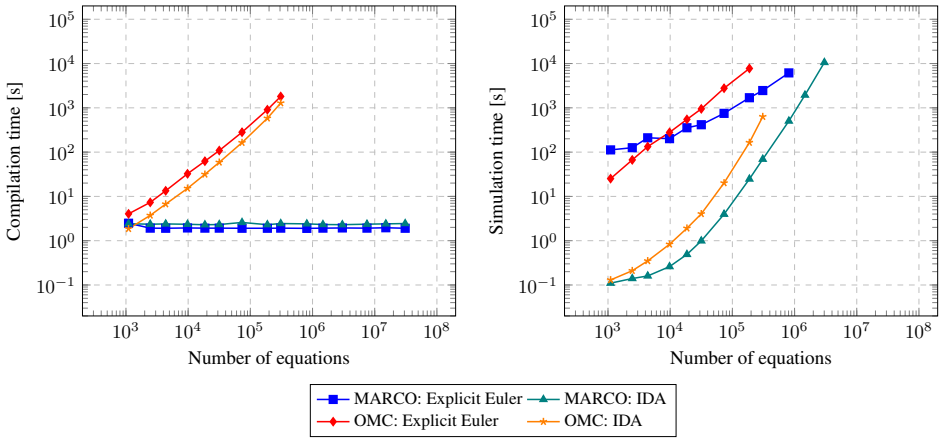

Figure 7.5: *Benchmark results for the silicon chip model with cooling channels (`ThermalChipCoolingOO`): log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

`ThermalChipOO` model (i.e., without cooling channels), with MARCO still maintaining a constant-time scaling behavior and OMC exhibiting a linear trend. In the case of explicit Euler, the simulation times of MARCO are approximately two orders of magnitude higher than those of the basic `ThermalChipOO` model, and initially higher than the ones of OMC, which are unvaried. Increasing the number of equations however demonstrates how, with respect to scalability, the approach and the algorithms followed by MARCO represent the most effective solution. Indeed, at 187911 equations – the largest model simulated by both the tools using the explicit Euler method – MARCO is able to complete the simulation in 1690.69 seconds, while OMC requires 7747.71 seconds. Moreover, when using the IDA solver, MARCO has faster simulation times since the smallest simulated model, and remains consistently faster than OMC as the number of equations increases. The inefficiency of the explicit Euler solution can therefore

				MARCO			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	1101	2.48	112.33	2.41	0.11
6	6	4	2457	1.92	125.81	2.36	0.14
8	8	4	4353	1.91	210.53	2.39	0.16
12	12	4	9765	1.94	201.38	2.35	0.26
15	15	5	18 594	1.92	355.05	2.29	0.49
18	18	6	31 653	1.92	415.35	2.32	0.99
24	24	8	73 521	1.91	750.83	2.58	3.96
33	33	11	187 911	1.91	1690.69	2.30	24.66
39	39	13	308 070	1.92	2458.03	2.44	69.21
54	54	18	809 361	1.90	6151.60	2.40	502.54
66	66	22	1 470 357	1.92	$>$ 3 h	2.32	1939.19
84	84	28	3 016 701	1.94	$>$ 3 h	2.30	10 579.3
114	114	38	7 505 541	1.92	$>$ 3 h	2.36	$>$ 3 h
144	144	48	15 085 881	1.96	$>$ 3 h	2.4	$>$ 3 h
183	183	61	30 893 886	1.91	$>$ 3 h	2.44	$>$ 3 h

				OMC			
				Explicit Euler		IDA	
N	M	P	Equations	Compilation	Simulation	Compilation	Simulation
4	4	4	1101	4.04 s	25.21 s	1.86 s	0.13 s
6	6	4	2457	7.34 s	66.75 s	3.73 s	0.21 s
8	8	4	4353	13.4 s	133.20 s	6.71 s	0.35 s
12	12	4	9765	32.47 s	279.05 s	15.18 s	0.83 s
15	15	5	18 594	62.38 s	548.08 s	31.52 s	1.91 s
18	18	6	31 653	108.56 s	947.96 s	58.83 s	4.04 s
24	24	8	73 521	281.21 s	2770.05 s	163.80 s	20.13 s
33	33	11	187 911	907.09 s	7747.71 s	582.05 s	164.31 s
39	39	13	308 070	1806.87 s	$>$ 3 h	1284.08 s	631.93 s
54	54	18	809 361	$>$ 1 h	-	$>$ 1 h	-

Table 7.5: *Benchmark results for the silicon chip model with cooling channels (*`ThermalChipCoolingOO`*): compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *ms. The stop time is set to* 40 *s for both explicit Euler and IDA.*

be attributed to the different configuration of the KINSOL solver (e.g., required accuracy), which is the only implementative difference between the two simulations for what regards the handling of the non-linear systems characterizing this model.

Heat Exchangers Model As discussed in Section 2.5, the Modelica language supports user-defined functions that enable the specification of arbitrary computations within a model. In the context of MARCO, support for such functions was initially introduced in 2021, during the first attempt – by the present author – to provide an MLIR-based implementation of a Modelica compiler. Since then, the multi-level IR of MARCO, as well as its runtime environment, integration with LLVM, and upstream transformations, have been significantly improved. It is therefore instructive to assess the performance of the current implementation relative to the original work, employing the same benchmark model as in the initial evaluation.

The model consists of a set of heat exchangers, organized into `Nu` groups connected in series. Each group contains `Nh` heat exchangers arranged in parallel, with a mixer linking each group to the subsequent one in the series. Furthermore, each heat exchanger is subdivided into `Nv` discrete volumes. Some methanol enters the system at a prescribed "cold" temperature and, after traversing all units in series, exits at a higher temperature. The specific enthalpy and specific heat capacity of the fluid are computed using the correlations proposed by Craven and De Reuck [21], under the assumption of incompressibility. Each heat exchanger is heated by a thermal power source that is equally distributed along the pipe. All these power sources are composed of a constant component and a sinusoidal component, with a different phase for each heat exchanger. At the end of each group there is a finite volume mixer, which gathers the methanol from the outlets of the heat exchangers and directs it to the inlets of the subsequent group.

The time measurements are reported in Table 7.6 and visualized in Figure 7.6. Com-

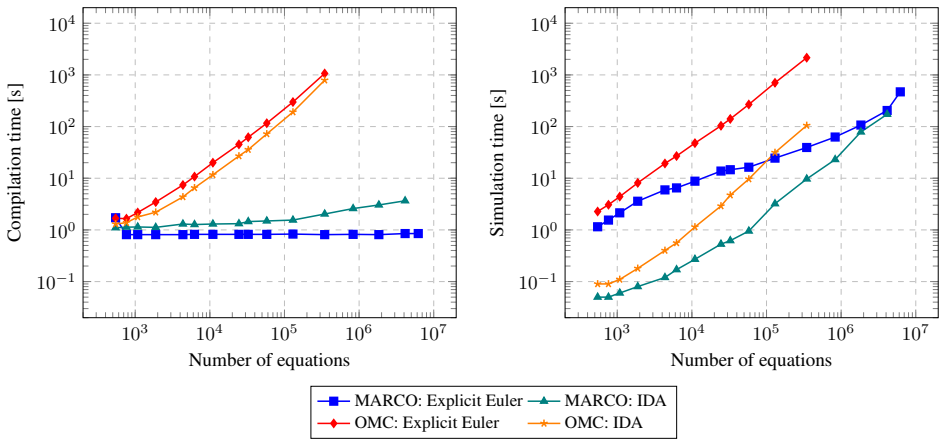

Figure 7.6: *Benchmark results for the heat exchangers model (*`MethanolHeatExchangersDAE`*): log-log plots of compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations.*

pilation times for the explicit Euler method remain constant across all model configurations, thereby confirming the scalability results observed in the original 2021 study. When employing the IDA solver, which was not available in the former work, compila-

				MARCO			
				Explicit Euler		IDA	
Nu	Nh	Nv	Equations	Compilation	Simulation	Compilation	Simulation
3	4	6	546	1.72 s	1.15 s	1.10 s	0.05 s
4	5	5	763	0.81 s	1.55 s	1.13 s	0.05 s
4	6	6	1080	0.81 s	2.14 s	1.14 s	0.06 s
4	8	8	1882	0.81 s	3.59 s	1.12 s	0.08 s
6	10	10	4356	0.81 s	5.95 s	1.30 s	0.12 s
6	12	12	6230	0.82 s	6.50 s	1.27 s	0.17 s
6	16	16	10 986	0.82 s	8.77 s	1.30 s	0.27 s
6	24	24	24 530	0.82 s	13.77 s	1.32 s	0.53 s
8	24	24	32 698	0.82 s	14.61 s	1.45 s	0.62 s
8	32	32	57 922	0.82 s	16.35 s	1.49 s	0.95 s
8	48	48	129 874	0.83 s	24.49 s	1.55 s	3.21 s
12	64	64	345 714	0.81 s	39.37 s	2.03 s	9.68 s
16	86	86	831 256	0.82 s	62.74 s	2.59 s	23.00 s
16	128	128	1 839 298	0.81 s	106.98 s	3.04 s	78.64 s
16	192	192	4 135 170	0.85 s	204.65 s	3.66 s	172.85 s

				OMC			
				Explicit Euler		IDA	
Nu	Nh	Nv	Equations	Compilation	Simulation	Compilation	Simulation
3	4	6	546	1.68 s	2.27 s	1.30 s	0.09 s
4	5	5	763	1.65 s	3.07 s	1.38 s	0.09 s
4	6	6	1080	2.17 s	4.41 s	1.78 s	0.11 s
4	8	8	1882	3.46 s	8.12 s	2.20 s	0.18 s
6	10	10	4356	7.40 s	19.31 s	4.35 s	0.40 s
6	12	12	6230	10.77 s	26.88 s	6.52 s	0.56 s
6	16	16	10 986	20.00 s	47.79 s	11.66 s	1.14 s
6	24	24	24 530	45.21 s	103.65 s	26.93 s	2.90 s
8	24	24	32 698	62.13 s	140.50 s	35.81 s	4.71 s
8	32	32	57 922	116.14 s	267.80 s	72.05 s	9.65 s
8	48	48	129 874	298.55 s	704.39 s	191.11 s	31.44 s
12	64	64	345 714	1066.14 s	2151.51 s	789.77 s	105.03 s
16	86	86	831 256	> 1 h	-	> 1 h	-

Table 7.6: *Benchmark results for the heat exchangers model (*`MethanolHeatExchangersDAE`*): compilation and simulation times for MARCO and OMC using the Explicit Euler numerical integration method and the IDA solver, as a function of the number of equations. The fixed time step of the explicit Euler method is set to* 1.5 *ms. The stop time is set to* 40 *s for both explicit Euler and IDA.*

tion times exhibit a modest linear increase due to the algebraic manipulations performed by the Clever DAE optimization; nevertheless, even in this scenario, the compilation performance of MARCO remains significantly superior to that of OMC. In terms of simulation times, MARCO consistently outperforms OMC in both the explicit Euler and IDA cases.

CHAPTER 8

Conclusions

THIS thesis addressed the challenge of efficiently compiling and simulating large-scale, array-based DAE systems by combining theoretical advancements with practical implementations.

On the theoretical front, novel data structures were developed to compactly represent arbitrarily high-dimensional sets of points, complemented by algorithms that recognize and accelerate common access patterns in array-based DAE systems while preserving general applicability. These theoretical contributions found their realization in the MARCO compiler through the adoption of the MLIR framework, which preserves the high-level semantics of the Base Modelica language throughout the compilation pipeline. This design choice enabled not only the application of the proposed algorithms but also a seamless integration with the runtime environment, which was engineered from the ground up to provide a modular and extensible architecture capable of exploiting parallel execution opportunities on modern multi-core processors.

This chapter analyzes how these theoretical and practical contributions address the research questions posed in Section 1.3 and identifies promising directions for future research.

8.1 Answer to Research Questions

This section revisits the research questions introduced in Section 1.3, and summarizes how the contributions of this thesis address each one through both theoretical innovation and numeric validation.

RQ1: How can multidimensional arrays be efficiently represented and manipulated during the compilation of DAE-based modeling languages?

Chapter 3 introduced compact representations for arbitrarily high-dimensional index spaces, accompanied by algorithms that leverage their properties throughout the causalization process.

These compact representations enable efficient querying of index space properties such as intersection and subtraction operations. As the number of scalar points grows, the *Range* and *MultidimensionalRange* classes maintain constant-time complexity for common operations, while the *IndexSet* class achieves logarithmic-time scaling through its underlying R-Tree implementation.

Building upon these foundations, specialized access pattern recognition (e.g., direct indexing, affine transformations) enables efficient mapping between equation and variable indices. To support the matching algorithm proposed by Fioravanti et al. [26], a compact incidence matrix was introduced. This was complemented by array-aware algorithms for SCC detection and scheduling, completing the causalization pipeline.

RQ2: Which changes within the compiler domain are required to effectively apply array-aware algorithms and improve the quality of code generation?

Chapter 4 introduced a multi-level IR design based on the MLIR framework, which Chapter 5 then elaborated through the implementation of a complete array-aware compilation pipeline.

This contribution brings several interconnected benefits. The multi-level design enables clear separation of concerns, with each dialect addressing distinct aspects of the compilation and simulation process. The `bmodelica` dialect captures DAE-specific constructs, the `ida`, `kinsol`, and `sundials` dialects decouple numerical solver logic from high-level model representation, and the `runtime` dialect maps directly to the APIs of the runtime libraries.

Beyond modularity, this multi-level approach enhances maintainability through individually testable dialect semantics and preserves array semantics throughout the compilation process — a crucial requirement for applying the array-aware algorithms developed in response to RQ1. At the same time, MLIR's extensible nature enables the reuse of existing dialects (e.g., `arith`, `scf`, etc.) for many compilation tasks, streamlining implementation and accelerating experimentation.

These architectural benefits position the compiler for future enhancements that would prove significantly more challenging within monolithic architectures.

RQ3: How can the runtime execution of DAE-based simulations be optimized to exploit parallel hardware architectures?

This question is addressed through the co-design of the compiler and runtime environment. The compiler identifies independent equations suitable for parallel execution and represents them as dedicated parallel execution regions in the generated code. Both the computational kernels and their associated metadata (iteration boundaries, dependencies) are lowered to predefined functions and global variables accessible by the runtime environment. The runtime infrastructure then partitions the iteration space and executes

these computations concurrently across available processor cores, effectively translating compiler-identified parallelism into actual performance gains.

8.2 Performance Achievements

The combined contributions addressing RQ1, RQ2, and RQ3 – compact index space representations, multi-level IR architecture, and parallel runtime capabilities – collectively enable MARCO to achieve substantial performance improvements over traditional scalar-expansion-based approaches.

Chapter 7 provided empirical validation of these achievements through benchmark models that are representative of large-scale, array-based thermal modeling scenarios. The assessment compared MARCO against the industry-grade and open-source OpenModelica compiler (OMC), focusing on both compilation and simulation performance.

The experimental results demonstrated that MARCO achieves near-constant compilation times as model size increases, contrasting sharply with traditional approaches that exhibit linear or worse scaling behaviors. For instance, in the case of object-oriented modeling of a silicon chip, MARCO compiled models with over 30 million scalar equations in approximately 2 seconds — the same time required for smaller versions of the same model. In contrast, OpenModelica exhibited linear scaling, requiring up to ~ 30 minutes of compilation time for models with $\sim 300\,000$ equations.

These results provide clear evidence that the compact array representations introduced in response to RQ1, combined with the multi-level IR architecture from RQ2, fundamentally change the scalability characteristics of DAE compilation. The ability to process models at this scale opens new possibilities for detailed thermal modeling and other application domains that were previously computationally infeasible.

Beyond compilation efficiency, simulation experiments demonstrated that the runtime environment efficiently exploits modern multi-core processors, yielding simulation times that are consistently on par with or superior to those of existing tools.

Together, these improvements in both compilation and simulation performance validate the thesis that array-aware compilation, when properly integrated with modern compiler infrastructure and parallel runtime support, can overcome the fundamental scalability limitations that have historically constrained large-scale DAE modeling.

8.3 Future Works

The foundations established in this thesis open several promising avenues for future research. These opportunities range from near-term enhancements that build directly upon the existing implementation to longer-term investigations that could significantly expand both the scope and impact of this work.

GPU and Accelerator Support

Perhaps the most impactful near-term enhancement involves extending both the compiler and runtime environment to support Graphics Processing Units (GPUs) and other hardware accelerators. Modern GPUs provide massive parallelism that could dramatically accelerate simulations of models originally described through PDEs, whose discretization methods naturally expose several parallelization opportunities. The MLIR

infrastructure is well-positioned for this extension, already incorporating dialects for GPU programming such as the `gpu` dialect and vendor-specific dialects like `nvvm` and `rocdl`. The primary challenge lies in identifying computations truly suitable for offloading based on both computational intensity and memory access patterns, alongside developing cost models capable of accurately evaluating the inherent trade-offs.

Cost-aware Scheduling and Online Tuning

Even within CPU-based execution, significant optimization opportunities remain unexplored. Currently, the parallel execution of independent equations relies on relatively simple heuristics that do not account for target architecture characteristics. Introducing architecture-aware cost models would enable more informed scheduling decisions, reducing idle thread times and memory pressure. Such optimizations apply not only during compilation but also at runtime, where online techniques could dynamically adapt execution parameters (e.g., switching between more coarse and fine-grained partitioning of the equations) to compensate for uncaptured costs or environmental variability.

Hybrid System Support

A critical limitation for widespread adoption of the current implementation is the absence of support for time and state events, which are essential for simulating hybrid systems combining continuous and discrete dynamics. Adding event handling capabilities would significantly broaden MARCO's applicability to domains such as control systems incorporating digital components. This extension requires augmenting the MLIR dialects to represent event-triggering conditions and implementing corresponding event handling mechanisms at the runtime level.

Full MLIR-Based Model Instantiation and Flattening

Currently, MARCO relies on OpenModelica's frontend for model instantiation (i.e., removing object orientation) and flattening (i.e., converting `connect` statements into equations). A significant long-term research direction involves implementing the entire compilation pipeline within the MLIR environment.

This would provide complete control over the compilation process from high-level Modelica code to executable binaries, while preserving the mid-level Base Modelica entry point for tools requiring Modelica-agnostic integration. Moreover, this independence would facilitate experimentation with Modelica language extensions without requiring coordination with third-party tools.

Extension to Partial Differential Equations

The proposed MLIR dialects could be extended to support PDEs and their spatial discretization methods. While neither Base Modelica nor Modelica currently support this level of abstraction, such an extension would align with the language experimentation theme and could enable support for other high-level modeling languages that incorporate PDE constructs [9].

Bibliography

[1] "A Conceptual Model for Digital Shadows in Industry and Its Application". In: *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2021, pp. 271–281. ISBN: 978-3-030-89021-6 978-3-030-89022-3. DOI: 10.1007/978-3-030-89022-3_22.

[2] Karim Abdelhak, Francesco Casella, and Bernhard Bachmann. "Pseudo Array Causalization". In: *15th International Modelica Conference 2023, Aachen, October 9-11*. Dec. 2023, pp. 177–188. DOI: 10.3384/ecp204177.

[3] Heimo H. Adelsberg. "Prolog as a Simulation Language". In: *Proceedings of the 16th Conference on Winter Simulation*. WSC '84. Dallas, TX: IEEE Press, 1984, pp. 500–504.

[4] Giovanni Agosta et al. "MARCO: An Experimental High-Performance Compiler for Large-Scale Modelica Models". In: *Proceedings of the 15th International Modelica Conference*. Linköping Electronic Conference Proceedings 204. Aachen, Dec. 2023. ISBN: 978-91-8075-505-4. DOI: 10.3384/ecp20413.

[5] Giovanni Agosta et al. "Towards a Benchmark Suite for High-Performance Modelica Compilers". In: *Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools*. Berlin Germany: ACM, Nov. 2019, pp. 21–24. ISBN: 978-1-4503-7713-3. DOI: 10.1145/3365984.3365988.

[6] Giovanni Agosta et al. "Towards a High-Performance Modelica Compiler". In: *The 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*. Feb. 2019, pp. 313–320. DOI: 10.3384/ecp19157313.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reprinted, with corr., [36. Druck]. Addison-Wesley Series in Computer Science. Reading, Mass.: Addison-Wesley, 2002. ISBN: 978-0-201-10088-4.

[8] D. J. Allerton. "The Impact of Flight Simulation in Aerospace". In: *The Aeronautical Journal* 114.1162 (2010), pp. 747–756. DOI: 10.1017/S0001924000004231.

[9] Carlos Amador, Mariano Martìn, and Laura Bueno. "Introduction to gPROMS for Chemical Engineering". In: *Introduction to Software for Chemical Engineers*. 2nd ed. 2019. ISBN: 978-0-429-45101-0.

[10] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. "A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications". In: *IEEE Access* 7 (2019), pp. 167653–167671. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2953499.

[11] J Bekker and W Lotz. "Planning Formula One Race Strategies Using Discrete-Event Simulation". In: *Journal of the Operational Research Society* 60.7 (July 2009), pp. 952–961. ISSN: 0160-5682, 1476-9360. DOI: 10.1057/palgrave.jors.2602626.

[12] Brahim Benhammouda. "A Novel Technique to Solve Nonlinear Higher-Index Hessenberg Differential - Algebraic Equations by Adomian Decomposition Method". In: *SpringerPlus* 5.1 (Dec. 2016). ISSN: 2193-1801. DOI: 10.1186/s40064-016-2208-3.

[13] Albert Benveniste, Benoit Caillaud, and Mathias Malandain. "The Mathematical Foundations of Physical Systems Modeling Languages". In: *Annual Reviews in Control* 50 (2020), pp. 72–118. ISSN: 13675788. DOI: 10.1016/j.arcontrol.2020.08.001.

[14] Louis G. Birta and Gilbert Arbez. *Modelling and Simulation: Exploring Dynamic System Behaviour*. Simulation Foundations, Methods and Applications. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-18868-9 978-3-030-18869-6. DOI: 10.1007/978-3-030-18869-6.

[15] Kathryn Eleda Brenan, Stephen L Campbell, and Linda Ruth Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. SIAM, 1995.

[16] Stephen L. Campbell and C. William Gear. "The Index of General Nonlinear DAEs". In: *Numerische Mathematik* 72.2 (Dec. 1995), pp. 173–196. ISSN: 0029-599X, 0945-3245. DOI: 10.1007/s002110050165.

[17] Francesco Casella. "Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives". In: *The 11th International Modelica Conference*. Sept. 2015, pp. 459–468. DOI: 10.3384/ecp15118459.

[18] François E. Cellier. "Combined Continuous/Discrete Simulation: Applications, Techniques and Tools". In: *Proceedings of the 18th Conference on Winter Simulation - WSC '86*. Washington, D.C., United States: ACM Press, 1986, pp. 24–33. DOI: 10.1145/318242.318256.

[19] François E. Cellier and Hilding Elmqvist. "Automated Formula Manipulation Supports Object-Oriented Continuous-System Modeling". In: *IEEE Control Systems* 13.2 (Apr. 1993), pp. 28–38. ISSN: 1066-033X, 1941-000X. DOI: 10.1109/37.206983.

[20] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, 2006. ISBN: 978-0-387-26102-7. DOI: 10.1007/0-387-30260-3.

[21] R. J. B. Craven and K. M. De Reuck. "Ideal-Gas and Saturation Properties of Methanol". In: *International Journal of Thermophysics* 7.3 (May 1986), pp. 541–552. ISSN: 0195-928X, 1572-9567. DOI: 10.1007/BF00502388.

[22] "Digital Twin Application to Energy Consumption Management in Production: A Literature Review". In: *IFIP Advances in Information and Communication Technology*. Cham: Springer Nature Switzerland, 2023, pp. 96–105. ISBN: 978-3-031-25181-8 978-3-031-25182-5. DOI: 10.1007/978-3-031-25182-5_10.

[23] Hilding Elmqvist et al. "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *Linköping Electronic Conference Proceedings*. Vol. 181. Linköping University Electronic Press, Sept. 2021, pp. 73–86. DOI: 10.3384/ecp2118173.

[24] "Finite Difference, Finite Element and Finite Volume Methods for Partial Differential Equations". In: *Handbook of Materials Modeling*. Dordrecht: Springer Netherlands, 2005, pp. 2415–2446. ISBN: 978-1-4020-3287-5 978-1-4020-3286-8. DOI: 10.1007/978-1-4020-3286-8_127.

[25] Massimo Fioravanti. "M.A.R.C.O. : An Experimental High-Performance Modelica Compiler for Large Scale Systems". MSc Thesis. Politecnico di Milano, Dec. 2020.

[26] Massimo Fioravanti et al. "Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models". In: *ACM Transactions on Mathematical Software* 49.3 (Sept. 2023), pp. 1–25. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3611661.

[27] Thomas Gierlinger, Daniel Danch, and André Stork. "Rendering Techniques for Mixed Reality". In: *Journal of Real-Time Image Processing* 5.2 (June 2010), pp. 109–120. ISSN: 1861-8200, 1861-8219. DOI: 10.1007/s11554-009-0137-x.

[28] Geoffrey Gordon. "The Development of the General Purpose Simulation System (GPSS)". In: *ACM SIGPLAN Notices* 13.8 (Aug. 1978), pp. 183–198. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/960118.808382.

[29] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[30] Adrien Guironnet et al. "Towards an Open-Source Solution Using Modelica for Time-Domain Simulation of Power Systems". In: *2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*. Sarajevo: IEEE, Oct. 2018, pp. 1–6. ISBN: 978-1-5386-4505-5. DOI: 10.1109/ISGTEurope.2018.8571872.

[31] Antonin Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *ACM SIGMOD Record* 14.2 (June 1984), pp. 47–57. ISSN: 0163-5808. DOI: 10.1145/971697.602266.

[32] Ernst Hairer, Michel Roche, and Christian Lubich. *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods*. Lecture Notes in Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989. ISBN: 978-3-540-51860-0 978-3-540-46832-5. DOI: 10.1007/bfb0093947.

[33] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II*. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. ISBN: 978-3-642-05220-0 978-3-642-05221-7. DOI: 10.1007/978-3-642-05221-7.

[34] Alexander Heilmeier, Michael Graf, and Markus Lienkamp. "A Race Simulation for Strategy Decisions in Circuit Motorsports". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. Maui, HI: IEEE, Nov. 2018, pp. 2986–2993. DOI: 10.1109/itsc.2018.8570012.

[35] Alan C. Hindmarsh et al. "SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers". In: *ACM Transactions on Mathematical Software* 31.3 (Sept. 2005), pp. 363–396. ISSN: 0098-3500, 1557-7295. DOI: 10.1145/1089014.1089020.

[36] Hrvoje Jasak. "OpenFOAM: Open Source CFD in Research and Industry". In: *International Journal of Naval Architecture and Ocean Engineering* 1.2 (Dec. 2009), pp. 89–94. ISSN: 2092-6782. DOI: 10.2478/ijnaoe-2013-0011.

[37] Melike Karta and Ercan Çelik. "On the Numerical Solution of Differential-Algebraic Equations with Hessenberg Index-3". In: *Discrete Dynamics in Nature and Society* 2012.1 (Jan. 2012). Ed. by Antonia Vecchio. ISSN: 1026-0226, 1607-887X. DOI: 10.1155/2012/147240.

[38] W. Kieffer, S. Moujaes, and N. Armbya. "CFD Study of Section Characteristics of Formula Mazda Race Car Wings". In: *Mathematical and Computer Modelling* 43.11-12 (June 2006), pp. 1275–1287. ISSN: 0895-7177. DOI: 10.1016/j.mcm.2005.03.011.

[39] Jyothsna Balakrishna Kodman, Balbir Singh, and Manikandan Murugaiah. "A Comprehensive Survey of Open-Source Tools for Computational Fluid Dynamics Analyses". In: *Journal of Advanced Research in Fluid Mechanics and Thermal Sciences* 119.2 (July 2024), pp. 123–148. ISSN: 2289-7879. DOI: 10.37934/arfmts.119.2.123148.

[40] Werner Kritzinger et al. "Digital Twin in Manufacturing: A Categorical Literature Review and Classification". In: *IFAC-PapersOnLine* 51.11 (2018), pp. 1016–1022. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2018.08.474.

[41] Gerd Kurzbach et al. "Design Proposal of a Standardized Base Modelica Language". In: *Linköping Electronic Conference Proceedings*. Vol. 204. Linköping University Electronic Press, Dec. 2023, pp. 469–478. DOI: 10.3384/ecp204469.

[42] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* San Jose, CA, USA: IEEE, Mar. 2004, pp. 75–86. DOI: 10.1109/cgo.2004.1281665.

[43] Chris Lattner et al. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Seoul, Korea (South): IEEE, Feb. 2021, pp. 2–14. ISBN: 978-1-7281-8613-9. DOI: 10.1109/CGO51591.2021.9370308.

[44] Averill M Law and W David Kelton. *Simulation Modeling and Analysis*. Vol. 3. Mcgraw-hill New York, 2000.

[45] G. C. Layek. *An Introduction to Dynamical Systems and Chaos*. University Texts in the Mathematical Sciences. Singapore: Springer Nature Singapore, 2024. ISBN: 978-981-99-7694-2 978-981-99-7695-9. DOI: 10.1007/978-981-99-7695-9.

[46] Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. "A State-of-the-Art Survey of Digital Twin: Techniques, Engineering Product Lifecycle Management and Business Innovation Perspectives". In: *Journal of Intelligent Manufacturing* 31.6 (Aug. 2020), pp. 1313–1337. ISSN: 0956-5515, 1572-8145. DOI: 10.1007/s10845-019-01512-w.

[47] D. J. N. Limebeer et al. "Optimal Control of a NASCAR - Specification Race Car". In: *Vehicle System Dynamics* 61.5 (May 2022), pp. 1210–1235. DOI: 10.1080/00423114.2022.2067573.

[48] Mengnan Liu et al. "Review of Digital Twin about Concepts, Technologies, and Industrial Applications". In: *Journal of Manufacturing Systems* 58 (Jan. 2021), pp. 346–361. ISSN: 02786125. DOI: 10.1016/j.jmsy.2020.06.017.

[49] Yingbo Ma et al. *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling*. Feb. 2022. DOI: 10.48550/arXiv.2103.05244. arXiv: 2103.05244 [cs].

[50] Mori Mani and Andrew J. Dorgan. "A Perspective on the State of Aerospace Computational Fluid Dynamics Technology". In: *Annual Review of Fluid Mechanics* 55.1 (Jan. 2023), pp. 431–457. ISSN: 0066-4189, 1545-4479. DOI: 10.1146/annurev-fluid-120720-124800.

[51] Vincenzo Manzoni and Francesco Casella. "Minimal Equation Sets for Output Computation in Object-Oriented Models". In: *The 8th International Modelica Conference, Technical Univeristy, Dresden, Germany*. June 2011, pp. 784–790. DOI: 10.3384/ecp11063784.

[52] Marek Matejak and Jiri Kofranek. "Physiomodel - an Integrative Physiology in Modelica". In: *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. Milan: IEEE, Aug. 2015, pp. 1464–1467. ISBN: 978-1-4244-9271-8. DOI: 10.1109/EMBC.2015.7318646.

[53] *Modelica: Language Specification*. https://specification.modelica.org/.

[54] Richard D. Neidinger. "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming". In: *SIAM Review* 52.3 (Jan. 2010), pp. 545–563. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/080743627.

[55] Constantinos C. Pantelides. "The Consistent Initialization of Differential-Algebraic Systems". In: *SIAM J. Sci. Stat. Comput.* 9.2 (1988), pp. 213–231. ISSN: 0196-5204.

[56] L. R. Petzold. "A Description of DASSL: A Differential/Algebraic System Solver". In: *10th International Mathematics and Computers Simulation Congress on Systems Simulation and Scientific Computation*. Montreal, Canada: Sandia National Labs., Livermore, CA (USA), 1982.

[57] Linda Petzold. "Differential/Algebraic Equations Are Not ODE' S". In: *SIAM Journal on Scientific and Statistical Computing* 3.3 (Sept. 1982), pp. 367–384. ISSN: 0196-5204, 2168-3417. DOI: 10.1137/0903023.

[58] Adrian Pop et al. "A New OpenModelica Compiler High Performance Frontend". In: *The 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*. Feb. 2019, pp. 689–698. DOI: 10.3384/ecp19157689.

[59] Alfio Quarteroni and Alberto Valli. *Numerical Approximation of Partial Differential Equations*. Springer Series in Computational Mathematics. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. ISBN: 978-3-540-85267-4 978-3-540-85268-1. DOI: 10.1007/978-3-540-85268-1.

[60] Sylvain Quoilin et al. "ThermoCycle: A Modelica Library for the Simulation of Thermodynamic Systems". In: *The 10th International Modelica Conference, March 10-12, 2014, Lund, Sweden*. Mar. 2014, pp. 683–692. DOI: 10.3384/ecp14096683.

[61] Christopher Rackauckas and Qing Nie. "DifferentialEquations.Jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *Journal of open research software* 5.1 (May 2017), p. 15. DOI: 10.5334/jors.151.

[62] Michele Scuttari. "Design and Implementation of a Modelica Compiler with MLIR and LLVM". MSc Thesis. Politecnico di Milano, Oct. 7, 2021.

[63] Michele Scuttari et al. "Clever DAE: Compiler Optimizations for Digital Twins at Scale". In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. Bologna, Italy: ACM, May 2023, pp. 344–350. ISBN: 979-8-4007-0140-5. DOI: 10.1145/3587135.3589945.

[64] Samad M. E. Sepasgozar. "Differentiating Digital Twin from Digital Shadow: Elucidating a Paradigm Shift to Expedite a Smart, Sustainable Built Environment". In: *Buildings* 11.4 (Apr. 2021), p. 151. ISSN: 2075-5309. DOI: 10.3390/buildings11040151.

[65] Robert E Shannon. "Introduction to Simulation Languages". In: *Winter Simulation Conference*. Vol. 77. 1977, pp. 14–20.

[66] J. G. Shanthikumar and R. G. Sargent. "A Unifying View of Hybrid Simulation/Analytic Models and Modeling". In: *Operations Research* 31.6 (Dec. 1983), pp. 1030–1052. ISSN: 0030-364X, 1526-5463. DOI: 10.1287/opre.31.6.1030.

[67] Magnus Själander et al. *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. July 2024. DOI: 10.48550/arXiv.1912.05848. arXiv: 1912.05848 [cs].

[68] Martin Sjarov et al. "The Digital Twin Concept in Industry - A Review and Systematization". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2020, pp. 1789–1796. DOI: 10.1109/ETFA46521.2020.9212089.

[69] Massimo Stellato, Luca Bergianti, and Alessandro Picarelli. "Race Car Cooling System Model for Real Time Use in a Driving Simulator". In: *Linköping Electronic Conference Proceedings*. Vol. 204. Linköping University Electronic Press, Dec. 2023, pp. 675–682. DOI: 10.3384/ecp204675.

[70] Akatsuka Takeaki, Kato Akinori, and Yoshida Nobuyuki. "Integrated Continuous and Discrete System Simulation Program (CDSP)". In: *Information Processing in Japan* 13 (Jan. 1973), pp. 23–28.

[71] Federico Terraneo et al. "3D-ICE 3.0: Efficient Nonlinear MPSoC Thermal Simulation With Pluggable Heat Sink Models". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.4 (Apr. 2022), pp. 1062–1075. ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2021.3074613.

[72] L. Vanfretti et al. "iTesla Power Systems Library (iPSL): A Modelica Library for Phasor Time-Domain Simulations". In: *SoftwareX* 5 (2016), pp. 84–88. ISSN: 23527110. DOI: 10.1016/j.softx.2016.05.001.

[73] Jelena Vidakovic et al. "Flight Simulation Training Devices: Application, Classification, and Research". In: *International Journal of Aeronautical and Space Sciences* 22.4 (Aug. 2021), pp. 874–885. ISSN: 2093-2480. DOI: 10.1007/s42405-021-00358-y.

[74] R.C. Vieira and E.C. Biscaia. "Direct Methods for Consistent Initialization of DAE Systems". In: *Computers & Chemical Engineering* 25.9-10 (Sept. 2001), pp. 1299–1311. ISSN: 00981354. DOI: 10.1016/S0098-1354(01)00702-5.

[75] Michael Wetter et al. "Modelica Buildings Library". In: *Journal of Building Performance Simulation* 7.4 (July 2014), pp. 253–270. ISSN: 1940-1493, 1940-1507. DOI: 10.1080/19401493.2013.765506.

[76] Bernard P. Zeigler and Alexander Muzy. "From Discrete Event Simulation to Discrete Event Specified Systems (DEVS)". In: *IFAC-PapersOnLine* 50.1 (July 2017), pp. 3039–3044. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.672.

APPENDIX A

Benchmarks

THIS appendix reports the source code for the models used in Chapter 7 to evaluate the performance of MARCO and of the generated simulations.

A.1 Wire

```
model Wire
   parameter Real g = 0.00314785;
   parameter Real c = 0.2707936;
   parameter Integer N = 10;
   Real[N] T;
   Real T1;
   Real TN;
equation
   for i in 2 : N - 1 loop
      c * der(T[i]) = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i]);
   end for;

   c * der(T[1]) = g * (T[2] - T[1]) + 2 * g * (673.15 - T[1]);
   c * der(T[N]) = g * (T[N - 1] - T[N]) + 2 * g * (293.15 - T[N]);
   T1 = T[1];
   TN = T[N];
initial equation
   for i in 1 : N loop
      T[i] = 293.15;
   end for;
end Wire;
```

A.2 Silicon Chip: ODE

```
package ThermalChipODE
  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 Interfaces
  end Interfaces;

  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" annotation(
        Evaluate = true);
      parameter Types.Length W = 12e-3 "Chip width in the y direction" annotation(
        Evaluate = true);
      parameter Types.Length H = 4e-3 "Chip height in the z direction" annotation(
        Evaluate = true);
      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon" annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
        Evaluate = true);
      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" annotation(
        Evaluate = true);
      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 Qb[N,M] "Power injected in the bottom volumes";

      output Types.Temperature Tct1 = T[1,1,1] "Top layer corner volume temperature";
      output Types.Temperature Tct2 = T[1,N,1] "Top layer corner volume temperature";
      output Types.Temperature Tct3 = T[N,N,1] "Top layer corner volume temperature";
      output Types.Temperature Tct4 = T[N,1,1] "Top layer corner volume temperature";
      output Types.Temperature Tcb1 = T[1,1,P] "Bottom layer corner volume temperature";
      output Types.Temperature Tcb2 = T[1,N,P] "Bottom layer corner volume temperature";
      output Types.Temperature Tcb3 = T[N,N,P] "Bottom layer corner volume temperature";
      output Types.Temperature Tcb4 = T[N,1,P] "Bottom layer corner volume temperature";
    equation
      der(T[1,1,1]) = 1/C*(Gx*((-T[1,1,1]) + T[2,1,1]) +
                      Gy*((-T[1,1,1]) + T[1,2,1]) +
                      Gz*(2*Tt-3*T[1,1,1] + T[1,1,2])) "Upper left top corner";

      der(T[N,1,1]) = 1/C*(Gx*(T[N-1,1,1]-T[N,1,1]) +
                      Gy*((-T[N,1,1]) + T[N,2,1]) +
                      Gz*(2*Tt-3*T[N,1,1] + T[N,1,2])) "Lower left top corner";

      der(T[1,M,1]) = 1/C*(Gx*((-T[1,M,1]) + T[2,M,1]) +
                      Gy*(T[1,M-1,1]-T[1,M,1]) +
                      Gz*(2*Tt-3*T[1,M,1] + T[1,M,2])) "Upper right top corner";

      der(T[N,M,1]) = 1/C*(Gx*(T[N-1,M,1]-T[N,M,1]) +
                      Gy*(T[N,M-1,1]-T[N,M,1]) +
                      Gz*(2*Tt-3*T[N,M,1] + T[N,M,2])) "Lower right top corner";

      der(T[1,1,P]) = 1/C*(Gx*((-T[1,1,P]) + T[2,1,P]) +
                      Gy*((-T[1,1,P]) + T[1,2,P]) +
                      Gz*(T[1,1,P-1]-T[1,1,P]) + Qb[1,1]) "Upper left bottom corner";

      der(T[N,1,P]) = 1/C*(Gx*(T[N-1,1,P]-T[N,1,P]) +
                      Gy*((-T[N,1,P]) + T[N,2,P]) +
                      Gz*(T[N,1,P-1]-T[N,1,P]) + Qb[N,1]) "Lower left bottom corner";

      der(T[1,M,P]) = 1/C*(Gx*((-T[1,M,P]) + T[2,M,P]) +
                      Gy*(T[1,M-1,P]-T[1,M,P]) +
                      Gz*(T[1,M,P-1]-T[1,M,P]) + Qb[1,M]) "Upper right bottom corner";

      der(T[N,M,P]) = 1/C*(Gx*(T[N-1,M,P]-T[N,M,P]) +
                      Gy*(T[N,M-1,P]-T[N,M,P]) +
                      Gz*(T[N,M,P-1]-T[N,M,P]) + Qb[N,M]) "Lower right bottom corner";
```

```
88
89      for i in 2:N-1 loop
90       der(T[i,1,1]) = 1/C*(Gx*(T[i-1,1,1]-2*T[i,1,1] + T[i+1,1,1]) +
91                       Gy*((-T[i,1,1]) + T[i,2,1]) +
92                       Gz*(2*Tt-3*T[i,1,1] + T[i,1,2])) "Left top edge";
93
94       der(T[i,M,1]) = 1/C*(Gx*(T[i-1,M,1]-2*T[i,M,1] + T[i+1,M,1]) +
95                       Gy*(T[i,M-1,1]-T[i,M,1]) +
96                       Gz*(2*Tt-3*T[i,M,1] + T[i,M,2])) "Right top edge";
97
98       der(T[i,1,P]) = 1/C*(Gx*(T[i-1,1,P]-2*T[i,1,P] + T[i+1,1,P]) +
99                       Gy*((-T[i,1,P]) + T[i,2,P]) +
100                      Gz*(T[i,1,P-1]-T[i,1,P]) + Qb[i,1]) "Left bottom edge";
101
102      der(T[i,M,P]) = 1/C*(Gx*(T[i-1,M,P]-2*T[i,M,P] + T[i+1,M,P]) +
103                      Gy*(T[i,M-1,P]-T[i,M,P]) +
104                      Gz*(T[i,M,P-1]-T[i,M,P]) + Qb[i,M]) "Right bottom edge";
105     end for;
106
107     for j in 2:M-1 loop
108      der(T[1,j,1]) = 1/C*(Gx*((-T[1,j,1]) + T[2,j,1]) +
109                      Gy*(T[1,j-1,1]-2*T[1,j,1] + T[1,j+1,1]) +
110                      Gz*(2*Tt-3*T[1,j,1] + T[1,j,2])) "Upper top edge";
111
112      der(T[N,j,1]) = 1/C*(Gx*(T[N-1,j,1]-T[N,j,1]) +
113                      Gy*(T[N,j-1,1]-2*T[N,j,1] + T[N,j+1,1]) +
114                      Gz*(2*Tt-3*T[N,j,1] + T[N,j,2])) "Lower top edge";
115
116      der(T[1,j,P]) = 1/C*(Gx*((-T[1,j,P]) + T[2,j,P]) +
117                      Gy*(T[1,j-1,P]-2*T[1,j,P] + T[1,j+1,P]) +
118                      Gz*(T[1,j,P-1]-T[1,j,P]) + Qb[1,j]) "Upper bottom edge";
119
120      der(T[N,j,P]) = 1/C*(Gx*(T[N-1,j,P]-T[N,j,P]) +
121                      Gy*(T[N,j-1,P]-2*T[N,j,P] + T[N,j+1,P]) +
122                      Gz*(T[N,j,P-1]-T[N,j,P]) + Qb[N,j]) "Lower bottom edge";
123     end for;
124
125     for k in 2:P-1 loop
126      der(T[1,1,k]) = 1/C*(Gx*((-T[1,1,k]) + T[2,1,k]) +
127                      Gy*((-T[1,1,k]) + T[1,2,k]) +
128                      Gz*(T[1,1,k-1]-2*T[1,1,k] + T[1,1,k + 1])) "Upper left edge";
129
130      der(T[N,1,k]) = 1/C*(Gx*(T[N-1,1,k]-T[N,1,k]) +
131                      Gy*((-T[N,1,k]) + T[N,2,k]) +
132                      Gz*(T[N,1,k-1]-2*T[N,1,k] + T[N,1,k + 1])) "Lower left edge";
133
134      der(T[1,M,k]) = 1/C*(Gx*(T[1,M-1,k]-T[1,M,k]) +
135                      Gy*(T[2,M,k]-T[1,M,k]) +
136                      Gz*(T[1,M,k-1]-2*T[1,M,k] + T[1,M,k + 1])) "Upper right edge";
137
138      der(T[N,M,k]) = 1/C*(Gx*(T[N-1,M,k]-T[N,M,k]) +
139                      Gy*(T[N,M-1,k]-T[N,M,k]) +
140                      Gz*(T[N,M,k-1]-2*T[N,M,k] + T[N,M,k + 1])) "Lower right edge";
141     end for;
142
143     for i in 2:N-1 loop
144      for j in 2:M-1 loop
145       der(T[i,j,1]) = 1/C*(Gx*(T[i-1,j,1]-2*T[i,j,1] + T[i+1,j,1]) +
146                       Gy*(T[i,j-1,1]-2*T[i,j,1] + T[i,j+1,1]) +
147                       Gz*(2*Tt-3*T[i,j,1] + T[i,j,2])) "Top face";
148
149       der(T[i,j,P]) = 1/C*(Gx*(T[i-1,j,P]-2*T[i,j,P] + T[i+1,j,P]) +
150                       Gy*(T[i,j-1,P]-2*T[i,j,P] + T[i,j+1,P]) +
151                       Gz*(T[i,j,P-1]-T[i,j,P]) + Qb[i,j]) "Bottom face";
152      end for;
153     end for;
154
155     for i in 2:N-1 loop
156      for k in 2:P-1 loop
157       der(T[i,1,k]) = 1/C*(Gx*(T[i-1,1,k]-2*T[i,1,k] + T[i+1,1,k]) +
158                       Gy*((-T[i,1,k]) + T[i,2,k]) +
159                       Gz*(T[i,1,k-1]-2*T[i,1,k] + T[i,1,k + 1])) "Left face";
160
161       der(T[i,M,k]) = 1/C*(Gx*(T[i-1,M,k]-2*T[i,M,k] + T[i+1,M,k]) +
162                       Gy*(T[i,M-1,k]-T[i,M,k]) +
163                       Gz*(T[i,M,k-1]-2*T[i,M,k] + T[i,M,k + 1])) "Right face";
164      end for;
165     end for;
166
167     for j in 2:M-1 loop
168      for k in 2:P-1 loop
```

```
        der(T[1,j,k]) = 1/C*(Gx*((-T[1,j,k]) + T[2,j,k]) +
                        Gy*(T[1,j-1,k]-2*T[1,j,k] + T[1,j+1,k]) +
                        Gz*(T[1,j,k-1]-2*T[1,j,k] + T[1,j,k + 1])) "Upper face";
        der(T[N,j,k]) = 1/C*(Gx*(T[N-1,j,k]-T[N,j,k]) +
                        Gy*(T[N,j-1,k]-2*T[N,j,k] + T[N,j+1,k]) +
                        Gz*(T[N,j,k-1]-2*T[N,j,k] + T[N,j,k + 1])) "Lower face";
      end for;
    end for;

    for i in 2:N-1 loop
      for j in 2:M-1 loop
        for k in 2:P-1 loop
          der(T[i,j,k]) = 1/C*(Gx*(T[i-1,j,k]-2*T[i,j,k] + T[i+1,j,k]) +
                          Gy*(T[i,j-1,k]-2*T[i,j,k] + T[i,j+1,k]) +
                          Gz*(T[i,j,k-1]-2*T[i,j,k] + T[i,j,k + 1])) "Internal volume";
        end for;
      end for;
    end for;
  end BaseThermalChip;

  model ThermalChipSimpleBoundary "Thermal chip model written by explicit ODEs, 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;

package Test
  model ThermalChipSimpleBoundary
    extends Models.ThermalChipSimpleBoundary;
  end ThermalChipSimpleBoundary;
end Test;
end ThermalChipODE;
```

A.3 Silicon Chip: DAE

```
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 Interfaces
  end Interfaces;

  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" annotation(
        Evaluate = true);
      parameter Types.Length W = 12e-3 "Chip width in the y direction" annotation(
        Evaluate = true);
      parameter Types.Length H = 4e-3 "Chip height in the z direction" annotation(
        Evaluate = true);
      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon" annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
        Evaluate = true);
      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";
```

```
37      final parameter Types.Length h = H / P "Chip height in the z direction";
38      parameter Types.Temperature Tt = 273.15 + 40 "Prescribed temperature of the top surface" annotation(
39        Evaluate = true);
40      final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal capacitance of a volume";
41      final parameter Types.ThermalConductance Gx = lambda*w*h / l "Thermal conductance of a volume,x direction";
42      final parameter Types.ThermalConductance Gy = lambda*l*h / w "Thermal conductance of a volume,y direction";
43      final parameter Types.ThermalConductance Gz = lambda*l*w / h "Thermal conductance of a volume,z direction";
44
45      Types.Temperature T[N,M,P](each start = Tstart,each fixed = true) "Temperatures of the volumes";
46      Types.Power Qx[N+1,M,P] "Power flows through the surfaces of the volumes, x direction";
47      Types.Power Qy[N,M+1,P] "Power flows through the surfaces of the volumes, x direction";
48      Types.Power Qz[N,M,P+1] "Power flows through the surfaces of the volumes, x direction";
49      Types.Power Qb[N,M] "Power injected in the bottom volumes";
50
51      output Types.Temperature Tct1 = T[1,1,1] "Top layer corner volume temperature";
52      output Types.Temperature Tct2 = T[1,N,1] "Top layer corner volume temperature";
53      output Types.Temperature Tct3 = T[N,N,1] "Top layer corner volume temperature";
54      output Types.Temperature Tct4 = T[N,1,1] "Top layer corner volume temperature";
55      output Types.Temperature Tcb1 = T[1,1,P] "Bottom layer corner volume temperature";
56      output Types.Temperature Tcb2 = T[1,N,P] "Bottom layer corner volume temperature";
57      output Types.Temperature Tcb3 = T[N,N,P] "Bottom layer corner volume temperature";
58      output Types.Temperature Tcb4 = T[N,1,P] "Bottom layer corner volume temperature";
59    equation
60      for i in 1:N loop
61        for j in 1:M loop
62          for k in 1:P loop
63            C*der(T[i,j,k]) = Qx[i, j,k] + Qy[i,j, k] + Qz[i,j,k ]
64                              -Qx[i+1,j,k] - Qy[i,j+1,k] - Qz[i,j,k+1] "Energy balance on each volume";
65          end for;
66        end for;
67      end for;
68      for j in 1:M loop
69        for k in 1:P loop
70          Qx[1,j,k] = 0;
71          for i in 2:N loop
72            Qx[i,j,k] = Gx*(T[i-1,j,k] - T[i,j,k]);
73          end for;
74          Qx[N+1,j,k] = 0;
75        end for;
76      end for;
77      for i in 1:N loop
78        for k in 1:P loop
79          Qy[i,1,k] = 0;
80          for j in 2:M loop
81            Qy[i,j,k] = Gy*(T[i,j-1,k] - T[i,j,k]);
82          end for;
83          Qy[i,M+1,k] = 0;
84        end for;
85      end for;
86      for i in 1:N loop
87        for j in 1:M loop
88          Qz[i,j,1] = 2*Gz*(Tt - T[i,j,1]);
89          for k in 2:P loop
90            Qz[i,j,k] = Gz*(T[i,j,k-1] - T[i,j,k]);
91          end for;
92          Qz[i,j,P+1] = -Qb[i,j];
93        end for;
94      end for;
95    end BaseThermalChip;
96
97    model ThermalChipSimpleBoundary "Thermal chip model written by explicit ODEs, constant power on half of the
          bottom surface"
98      extends BaseThermalChip;
99      parameter Types.Power Ptot = 100 "Total power consumption";
100     final parameter Types.Power Pv = Ptot / (N * M / 2) "Power dissipated in a single volume";
101   equation
102     Qb[:, 1:div(M, 2)] = fill(Pv, N, div(M, 2));
103     Qb[:, div(M, 2) + 1:end] = fill(0, N, M - div(M, 2));
104   end ThermalChipSimpleBoundary;
105 end Models;
106
107 package Test
108   model ThermalChipSimpleBoundary
109     extends Models.ThermalChipSimpleBoundary;
110   end ThermalChipSimpleBoundary;
111 end Test;
112 end ThermalChipDAE;
```

A.4 Silicon Chip: OO

```
package ThermalChipOO
  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");
    connector PowerInput = input Power;
  end Types;

  package Interfaces
    connector HeatPort
      Types.Temperature T;
      flow Types.Power Q;
    end HeatPort;
  end Interfaces;

  package Models

    model Volume
      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon" annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
        Evaluate = true);
      parameter Types.Temperature Tstart = 273.15 + 40;
      parameter Types.ThermalCapacitance C "Thermal capacitance of a volume";
      parameter Types.ThermalConductance Gx "Thermal conductance of half a volume,x direction";
      parameter Types.ThermalConductance Gy "Thermal conductance of half a volume,y direction";
      parameter Types.ThermalConductance Gz "Thermal conductance of half a volume,z
    direction";

      Interfaces.HeatPort upper "Upper surface thermal port";
      Interfaces.HeatPort lower "Lower surface thermal port";
      Interfaces.HeatPort left "Left surface thermal port";
      Interfaces.HeatPort right "Right surface thermal port";
      Interfaces.HeatPort top "Top surface thermal port";
      Interfaces.HeatPort bottom "Bottom surface thermal port";
      Interfaces.HeatPort center "Volume center thermal port";

      Types.Temperature T(start = Tstart, fixed = true) "Volume temperature";
    equation
      C*der(T) = upper.Q + lower.Q + left.Q + right.Q + top.Q + bottom.Q + center.Q;

      upper.Q = Gx*(upper.T - T);
      lower.Q = Gx*(lower.T - T);
      left.Q = Gy*(left.T - T);
      right.Q = Gy*(right.T - T);
      top.Q =  Gz*(top.T - T);
      bottom.Q = Gz*(bottom.T - T);
      center.T = T;
    end Volume;

    model TemperatureSource
      Interfaces.HeatPort port;
      Types.Temperature T = 298.15 "Source temperature";
    equation
  port.T = T;
    end TemperatureSource;

    model PowerSource
      Interfaces.HeatPort port;
      Types.PowerInput Q "Source thermal power leaving the port";
    equation
      port.Q = -Q;
    end PowerSource;
    partial model BaseThermalChip
      parameter Integer N = 15 "Number of volumesin the x direction";
      parameter Integer M = 15 "Number of volumesin the y direction";
      parameter Integer P = 5 "Number of volumesin the z direction";
      parameter Types.Length L = 12e-3 "Chip lengthin the x direction" annotation(
        Evaluate = true);
```

```
78      parameter Types.Length W = 12e-3 "Chip widthin the y direction" annotation(
79        Evaluate = true);
80      parameter Types.Length H = 4e-3 "Chip heightin the z direction" annotation(
81        Evaluate = true);
82      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
83        Evaluate = true);
84      parameter Types.Density rho = 2329 "Density of silicon" annotation(
85        Evaluate = true);
86      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
87        Evaluate = true);
88      parameter Types.Temperature Tstart = 273.15 + 40;
89      final parameter Types.Length l = L / N "Chip lengthin the x direction";
90      final parameter Types.Length w = W / M "Chip widthin the y direction";
91      final parameter Types.Length h = H / P "Chip heightin the z direction";
92      parameter Types.Temperature Tt = 273.15 + 40 "Prescribed temperature of the top surface" annotation(
93        Evaluate = true);
94      final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal capacitance of a volume";
95      final parameter Types.ThermalConductance Gx = lambda*w*h/l "Thermal conductance of a volume,x direction";
96      final parameter Types.ThermalConductance Gy = lambda*l*h/w "Thermal conductance of a volume,y direction";
97      final parameter Types.ThermalConductance Gz = lambda*l*w/h "Thermal conductance of a volume,z direction";
98
99      Volume vol[N,M,P](each T(start = Tstart, fixed = true),
100                       each C = C,
101                       each Gx = 2*Gx, each Gy = 2*Gy, each Gz = 2*Gz);
102     TemperatureSource Tsource[N,M](each T = Tt);
103
104     output Types.Temperature Tct1 = vol[1,1,1].T "Top layer corner volume temperature";
105     output Types.Temperature Tct2 = vol[1,N,1].T "Top layer corner volume temperature";
106     output Types.Temperature Tct3 = vol[N,N,1].T "Top layer corner volume temperature";
107     output Types.Temperature Tct4 = vol[N,1,1].T "Top layer corner volume temperature";
108     output Types.Temperature Tcb1 = vol[1,1,P].T "Bottom layer corner volume temperature";
109     output Types.Temperature Tcb2 = vol[1,N,P].T "Bottom layer corner volume temperature";
110     output Types.Temperature Tcb3 = vol[N,N,P].T "Bottom layer corner volume temperature";
111     output Types.Temperature Tcb4 = vol[N,1,P].T "Bottom layer corner volume temperature";
112   equation
113     // Connections in the z direction
114     for i in 1:N loop
115       for j in 1:M loop
116         connect(vol[i,j,1].top, Tsource[i,j].port);
117         for k in 1:P-1 loop
118           connect(vol[i,j,k].bottom, vol[i,j,k+1].top);
119         end for;
120       end for;
121     end for;
122
123     // Connections in the y direction
124     for i in 1:N loop
125       for k in 1:P loop
126         for j in 1:M-1 loop
127           connect(vol[i,j,k].right, vol[i,j+1,k].left);
128         end for;
129       end for;
130     end for;
131
132     // Connections in the x direction
133     for j in 1:M loop
134       for k in 1:P loop
135         for i in 1:N-1 loop
136           connect(vol[i,j,k].lower, vol[i+1,j,k].upper);
137         end for;
138       end for;
139     end for;
140   end BaseThermalChip;
141
142   model ThermalChipSimpleBoundary "Thermal chip model written by explicit ODEs, constant power on half of the
          bottom surface"
143     extends BaseThermalChip;
144     parameter Types.Power Ptot = 100 "Total power consumption";
145     final parameter Types.Power Pv = Ptot / (N * M / 2) "Power dissipated in a single volume";
146     PowerSource Qsource[N,div(M,2)](each Q = Pv);
147   equation
148     connect(Qsource.port, vol[:, 1:div(M,2), P].center);
149   end ThermalChipSimpleBoundary;
150 end Models;
151
152 package Test
153   model ThermalChipSimpleBoundary
154     extends Models.ThermalChipSimpleBoundary;
155   end ThermalChipSimpleBoundary;
156 end Test;
157 end ThermalChipOO;
```

A.5 Silicon Chip with Cooling Channels

```
package ThermalChipCoolingOO
  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 CoefficientOfHeatTransfer = Real(unit = "W/(m2.K)");
    type Density = Real(unit = "kg/m3");
    type MassFlowRate = Real(unit = "kg/s");
    type Length = Real(unit = "m");
    type Time = Real(unit = "s");
    type PerUnit = Real(unit = "1");
    connector PowerInput = input Power;
  end Types;

  package Interfaces
    connector HeatPort
      Types.Temperature T;
      flow Types.Power Q;
    end HeatPort;
  end Interfaces;

  package Models
    model Volume
      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Temperature Tstart = 273.15 + 40;
      parameter Types.ThermalCapacitance C "Thermal capacitance of a volume";
      parameter Types.ThermalConductance Gx "Thermal conductance of half a volume,x direction";
      parameter Types.ThermalConductance Gy "Thermal conductance of half a volume,y direction";
      parameter Types.ThermalConductance Gz "Thermal conductance of half a volume,z
         direction";
      Interfaces.HeatPort upper "Upper surface thermal port";
      Interfaces.HeatPort lower "Lower surface thermal port";
      Interfaces.HeatPort left "Left surface thermal port";
      Interfaces.HeatPort right "Right surface thermal port";
      Interfaces.HeatPort top "Top surface thermal port";
      Interfaces.HeatPort bottom "Bottom surface thermal port";
      Interfaces.HeatPort center "Volume center thermal port";
      Types.Temperature T(start = Tstart, fixed = true) "Volume temperature";
    equation
      C * der(T) = upper.Q + lower.Q + left.Q + right.Q + top.Q + bottom.Q + center.Q;
      upper.Q = Gx * (upper.T - T);
      lower.Q = Gx * (lower.T - T);
      left.Q = Gy * (left.T - T);
      right.Q = Gy * (right.T - T);
      top.Q = Gz * (top.T - T);
      bottom.Q = Gz * (bottom.T - T);
      center.T = T;
    end Volume;

    model TemperatureSource
      Interfaces.HeatPort port;
      Types.Temperature T = 298.15 "Source temperature";
    equation
      port.T = T;
    end TemperatureSource;

    model PowerSource
      Interfaces.HeatPort port;
      Types.PowerInput Q "Source thermal power leaving the port";
    equation
      port.Q = -Q;
    end PowerSource;

    partial model BaseThermalChip "Base model of fluid-cooled chip without power generation"
      parameter Integer N = 4 "Number of volumesin the x direction";
      parameter Integer M = 4 "Number of volumesin the y direction";
      parameter Integer P = 4 "Number of volumesin the z direction";
      parameter Types.Length L = 12e-3 "Chip lengthin the x direction" annotation(
```

```
73        Evaluate = true);
74      parameter Types.Length W = 12e-3 "Chip widthin the y direction" annotation(
75        Evaluate = true);
76      parameter Types.Length H = 4e-3 "Chip heightin the z direction" annotation(
77        Evaluate = true);
78      parameter Types.Length S = 1e-3 "Cooling channel thickness" annotation(
79        Evaluate = true);
80      parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation(
81        Evaluate = true);
82      parameter Types.Density rho = 2329 "Density of silicon" annotation(
83        Evaluate = true);
84      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
85        Evaluate = true);
86      parameter Types.Temperature Tstart = 273.15 + 25 "Start value of the temperature";
87      parameter Types.Temperature Tc = 273.15 + 25 "Inlet temperature of the cooling fluid";
88      parameter Types.Temperature Tt = 273.15 + 40 "Prescribed temperature of the top surface" annotation(
89        Evaluate = true);
90      parameter Types.CoefficientOfHeatTransfer gamma_n = 5e4 "Nominal heat transfer coefficient";
91      parameter Types.PerUnit alpha = 0.8 "Exponent of mass flow rate in heat transfer correlation";
92      parameter Types.MassFlowRate wn = 1e-3 "Nominal coolant mass flow rate";
93      final parameter Types.Length l = L / N "Chip length in the x direction";
94      final parameter Types.Length w = W / M "Chip width in the y direction";
95      final parameter Types.Length h = H / P "Chip height in the z direction";
96      final parameter Types.ThermalCapacitance C = rho * c * l * w * h "Thermal capacitance of a volume";
97      final parameter Types.ThermalConductance Gx = lambda * w * h / l "Thermal conductance of a volume,x direction
            ";
98      final parameter Types.ThermalConductance Gy = lambda * l * h / w "Thermal conductance of a volume,y direction
            ";
99      final parameter Types.ThermalConductance Gz = lambda * l * w / h "Thermal conductance of a volume,z direction
            ";
100     Volume vol[N, M, P](each T(start = Tstart, fixed = true), each C = C, each Gx = 2 * Gx, each Gy = 2 * Gy,
            each Gz = 2 * Gz);
101     //TemperatureSource Tsource[N, M](each T = Tt);
102     CoolingChannel cooling[M](each N = N, each L = L, each w = W / M, each S = S, each gamma_n = gamma_n, each
            alpha = alpha, each wn = wn / M, each T(each start = Tstart, each fixed = true));
103     Types.MassFlowRate wtot = 1e-3 * (1 + 0.1 * sin(time * 100));
104
105     output Types.Temperature Tct1 = vol[1, 1, 1].T "Top layer corner volume temperature";
106     output Types.Temperature Tct2 = vol[1, N, 1].T "Top layer corner volume temperature";
107     output Types.Temperature Tct3 = vol[N, N, 1].T "Top layer corner volume temperature";
108     output Types.Temperature Tct4 = vol[N, 1, 1].T "Top layer corner volume temperature";
109     output Types.Temperature Tcb1 = vol[1, 1, P].T "Bottom layer corner volume temperature";
110     output Types.Temperature Tcb2 = vol[1, N, P].T "Bottom layer corner volume temperature";
111     output Types.Temperature Tcb3 = vol[N, N, P].T "Bottom layer corner volume temperature";
112     output Types.Temperature Tcb4 = vol[N, 1, P].T "Bottom layer corner volume temperature";
113   equation
114 // Connections in the z direction
115     for i in 1:N loop
116       for j in 1:M loop
117         connect(vol[i, j, 1].top, cooling[j].heatPort[i]);
118         for k in 1:P - 1 loop
119           connect(vol[i, j, k].bottom, vol[i, j, k + 1].top);
120         end for;
121       end for;
122     end for;
123 // Connections in the y direction
124     for i in 1:N loop
125       for k in 1:P loop
126         for j in 1:M - 1 loop
127           connect(vol[i, j, k].right, vol[i, j + 1, k].left);
128         end for;
129       end for;
130     end for;
131 // Connections in the x direction
132     for j in 1:M loop
133       for k in 1:P loop
134         for i in 1:N - 1 loop
135           connect(vol[i, j, k].lower, vol[i + 1, j, k].upper);
136         end for;
137       end for;
138     end for;
139 // Cooling system boundaries
140     for j in 1:M loop
141       cooling[j].Ti = Tc;
142       cooling[j].wc = wtot * (1 + 0.1 * (j - 1 - (M - 1) / 2) / M) / M;
143     end for;
144   end BaseThermalChip;
145
146   model CoolingChannel
147     parameter Integer N = 2 "Number of volumes in the length direction";
148     parameter Types.SpecificHeatCapacity c = 4200 "Specific heat capacity of the cooling fluid";
```

```
    parameter Types.Density rho = 1000 "Density of the cooling fluid";
    parameter Types.Length L "Length of cooling channel";
    parameter Types.Length w "Width of cooling channel";
    parameter Types.Length S "Thickness of cooling channel";
    parameter Types.CoefficientOfHeatTransfer gamma_n "Nominal heat transfer coefficient";
    parameter Types.MassFlowRate wn "Nominal mass flow rate";
    parameter Types.PerUnit alpha "Exponent of mass flow rate in heat transfer coefficient";
    final parameter Types.Length l = L / N "Length of individual volume";
    Interfaces.HeatPort heatPort[N] "Heat port array";
    input Types.MassFlowRate wc "Mass flow rate through the channel";
    input Types.Temperature Ti "Inlet temperature of the fluid";
    Types.Power Q[N] "Thermal power entering each fluid volume";
    Types.Temperature T[N] "Outlet temperature of each fluid volume";
    Types.Temperature Ta[N] "Average temperature of each fluid volume";
    Types.Temperature Tw[N] "Average temperature of each volume wall";
    Types.CoefficientOfHeatTransfer gamma "Coefficient of heat transfer";
  equation
    rho * w * S * l * c * der(T[1]) = wc * c * (Ti - T[1]) + Q[1];
    Ta[1] = (Ti + T[1]) / 2;
    for i in 2:N loop
      rho * w * S * l * c * der(T[i]) = wc * c * (T[i - 1] - T[i]) + Q[i];
      Ta[i] = (T[i - 1] + T[i]) / 2;
    end for;
    gamma = (wc / wn) ^ alpha * gamma_n;
    for i in 1:N loop
      Q[i] = gamma * w * l * (Tw[i] - Ta[i]);
    end for;
    Q = heatPort.Q;
    Tw = heatPort.T;
  end CoolingChannel;

  model ThermalChipSimpleBoundary "Thermal chip model with cooling, 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";
    PowerSource Qsource[N, div(M, 2)](each Q = Pv);
  equation
    connect(Qsource.port, vol[:, 1:div(M, 2), P].center);
  end ThermalChipSimpleBoundary;
 end Models;

 package Test

model ThermalChipSimpleBoundary
 extends Models.ThermalChipSimpleBoundary;
end ThermalChipSimpleBoundary;
 end Test;
end ThermalChipCoolingOO;
```

A.6 Heat Exchangers

```
package MethanolHeatExchangersDAE
 package Types
   type Temperature = Real(unit = "K", displayUnit = "degC", nominal = 500);
   type Power = Real(unit = "W");
   type ThermalConductivity = Real(unit = "W/(m.K)");
   type ThermalConductance = Real(unit = "W/K");
   type SpecificEnthalpy = Real(unit = "J/kg");
   type SpecificHeatCapacity = Real(unit = "J/(kg.K)");
   type ThermalCapacitance = Real(unit = "J/K");
   type CoefficientOfHeatTransfer = Real(unit = "W/(m2.K)");
   type Density = Real(unit = "kg/m3");
   type MassFlowRate = Real(unit = "kg/s");
   type Length = Real(unit = "m");
   type Time = Real(unit = "s");
   type PerUnit = Real(unit = "1");
   type Frequency = Real(unit = "Hz");
   type Volume = Real(unit = "m3");
   type HeatCapacity = Real(unit = "J/K");
   type Pressure = Real(unit = "Pa");
 end Types;

 package Models
   model MethanolHeatExchangers
     parameter Integer Nu = 3 "Number of units in series";
```

```
25      parameter Integer Nh = 4 "Number of heat exchangers in parallel for each unit";
26      parameter Integer Nv = 6 "Number of finite volumes for each heat exchanger";
27      parameter Types.MassFlowRate w_nom = 1 "Nominal mass flow rate";
28      parameter Types.Power Q_nom = 500e3 "Nominal unit power";
29      parameter Types.Frequency f_w = 1/30 "Period of mass flow rate fluctuations";
30      parameter Types.Frequency f_Q = 1/100 "Period of thermal power fluctuations";
31      parameter Types.Temperature T0 = 493.15 "Reference temperature";
32      parameter Types.Volume V = 1 "Total volume of the fluid in the system";
33      parameter Types.PerUnit beta = 0.01 "Fraction of the total volume taken by mixing volumes";
34      parameter Types.ThermalConductance UA_nom = 10000 "Nominal UA value of each unit";
35      parameter Types.PerUnit alpha = 0.8 "Coefficient of mass flow rate in the UA value";
36      parameter Types.HeatCapacity Cw = 10000 "Total heat capacity of the heat exchanger walls of each unit";
37      parameter Types.Pressure p_nom = 20e5 "Nominal fluid pressure";
38      final parameter Types.Volume V_v = V*(1-beta)/(Nu*Nh*Nv) "Fluid volume of each finite volume";
39      final parameter Types.Volume V_m = V*beta/Nu;
40      final parameter Types.HeatCapacity C_wv = Cw/(Nu*Nh*Nv) "Heat capacity of the walls for each finite volume";
41      constant Real pi = 3.14159265359;
42      Types.MassFlowRate w "Total mass flow rate flowing through the system";
43      Types.MassFlowRate w_h "Mass flow rate flowing through each individual heat exchanger";
44      Types.Power Q[Nh] "Thermal power to each heat exchanger in a unit";
45      Types.Power Q_c[Nu,Nh,Nv] "Convective heat transfer on each heat exchanger volume";
46      Types.Temperature T[Nu,Nh,Nv+1] "Temperatures at the heat exchanger volume boundaries";
47      Types.SpecificEnthalpy h[Nu,Nh,Nv+1] "Specific enthalpies at the heat exchanger volume boundaries";
48      Types.SpecificEnthalpy h_m[Nu] "Specific enthalpies of the mixers at the unit outlets";
49      Types.Temperature T_tilde[Nu,Nh,Nv](each start = T0, each fixed = true) "Fluid temperature states";
50      Types.Density rho[Nu,Nh,Nv] "Density in each heat exchanger volume";
51      Types.Density rho_m[Nu] "Density in each mixer at the unit outlets";
52      Types.SpecificHeatCapacity cv[Nu,Nh,Nv] "cv in each heat exchanger volume";
53      Types.SpecificHeatCapacity cv_m[Nu] "cv in mixer at the unit outlets";
54      Types.Temperature T_w[Nu,Nh,Nv](each start = T0, each fixed = true) "Wall temperatures";
55      output Types.Temperature T_m[Nu](each start = T0, each fixed = true) "Temperatures of the mixers at each unit
            outlet";
56    equation
57      w = w_nom * (1 + 0.2*sin(2*pi*f_w*time));
58      w_h = w / Nh;
59      for j in 1:Nh loop
60        Q[j] = Q_nom/(Nu*Nh)*(1 + sin(2*pi*f_Q*time + 2*pi*j/Nh));
61      end for;
62      for j in 1:Nh loop
63        T[1,j,1] = T0 "Inlet temperatures of heat exchangers in first unit";
64      end for;
65      for i in 2:Nu loop
66        for j in 1:Nh loop
67          T[i,j,1] = T_m[i-1] "Inlet temperatures of heat exhangers of units 2:Nu";
68        end for;
69      end for;
70      T_tilde = T[:,:,2:Nv + 1] "Volume outlet temperatures";
71      for i in 1:Nu loop
72        // Energy balance equation for the mixers at the unit outlets
73        V_m*rho_m[i]*cv_m[i]*der(T_m[i]) = w_h*sum(h[i,j,Nv+1] for j in 1:Nh) - w*h_m[i];
74        for j in 1:Nh loop
75          for k in 1:Nv loop
76            // Energy balance equations for the heat exchangers
77            V_v*rho[i,j,k]*cv[i,j,k]*der(T_tilde[i,j,k]) = w_h*(h[i,j,k] - h[i,j,k+1]) + Q_c[i,j,k];
78            C_wv*der(T_w[i, j, k]) = Q[j]/Nv - Q_c[i,j,k];
79            Q_c[i,j,k] = UA_nom/(Nu*Nh*Nv)*(w/w_nom)^alpha*(T_w[i,j,k] - T_tilde[i,j,k]);
80          end for;
81        end for;
82      end for;
83    // Fluid property computations
84      for i in 1:Nu loop
85        rho_m[i] = p_nom /(Methanol.R*T_m[i]);
86        h_m[i] = Methanol.h_T(T_m[i]);
87        cv_m[i] = Methanol.cv_T(T_m[i]);
88        for j in 1:Nh loop
89          for k in 1:Nv loop
90            rho[i,j,k] = p_nom /(Methanol.R*T_tilde[i,j,k]);
91            cv[i,j,k] = Methanol.cv_T(T_tilde[i,j,k]);
92          end for;
93          for k in 1:Nv+1 loop
94            h[i,j,k] = Methanol.h_T(T[i,j,k]);
95          end for;
96        end for;
97      end for;
98      annotation(
99        experiment(StartTime = 0, StopTime = 100, Tolerance = 1e-06, Interval = 0.01),
100       __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "euler"));
101   end MethanolHeatExchangers;
102
103   package Methanol
104     constant Types.SpecificHeatCapacity R = 8.314462 / 32.04e-3;
```

```
105       constant Types.Temperature Tc = 512.64;
106       constant Types.PerUnit f[8] = {3.90086, 10.9929, 18.3371, -16.3663, -6.22334, 2.80358, 1.07783, 0.96967};
107       constant Types.PerUnit g[8] = {0.0, 4.12575, 3.26973, 3.77492, 2.93574, 8.23747, 10.3312, 0.53326};
108
109       function cp_T
110         input Types.Temperature T;
111         output Types.SpecificHeatCapacity cp;
112       protected
113         Types.PerUnit tau;
114         Types.PerUnit u[8];
115         Types.PerUnit x;
116       algorithm
117         tau := Tc / T;
118         u := g * tau;
119         x := f[1];
120         for i in 2:8 loop
121           x := x + f[i] * u[i] ^ 2 * exp(u[i]) / (exp(u[i]) - 1) ^ 2;
122         end for;
123         cp := x * R;
124       end cp_T;
125
126       function cv_T
127         input Types.Temperature T;
128         output Types.SpecificHeatCapacity cv;
129       algorithm
130         cv := cp_T(T) - R;
131       end cv_T;
132
133       function h_T
134         input Types.Temperature T;
135         output Types.SpecificEnthalpy h;
136       protected
137         Types.PerUnit tau;
138         Types.PerUnit u[8];
139         Types.PerUnit x;
140       algorithm
141         tau := Tc / T;
142         u := g * tau;
143         x := f[1] / tau;
144         for i in 2:8 loop
145           x := x + f[i] * g[i] / (exp(u[i]) - 1);
146         end for;
147         h := R * T * tau * x - 1361.810 * tau / Tc;
148       end h_T;
149     end Methanol;
150   end Models;
151 end MethanolHeatExchangersDAE;
```