

MLIR Dialects for DAE-based Modeling Languages

MICHELE SCUTTARI, Politecnico di Milano, Italy

DANIELE CATTANEO, Politecnico di Milano, Italy

FEDERICO TERRANEO, Politecnico di Milano, Italy

GIOVANNI AGOSTA, Politecnico di Milano, Italy

STEFANO CHERUBIN, Norwegian University of Science and Technology, Norway

Models based on Differential-Algebraic Equations (DAE) are ubiquitous across engineering fields. Specialized compilers and languages are key enablers to automate the numerical simulation of these models, which are experiencing a steady increase in their size. In this article, we discuss the uniqueness of this class of compilers, we propose a modern compiler design based on MLIR (Multi-Level Intermediate Representation), and we showcase its implementation – named MARCO (Modelica Advanced Research Compiler) – targeting Base Modelica, a simple yet industrially-relevant DAE-based language. The proposed design allows us to overcome the structural limitations of other modeling compilers. Our solution reduces the compilation time and the runtime complexities. We test our approach using two arbitrarily scalable thermal models implementing the finite volume spatial discretization method. On these benchmarks, MARCO achieves constant-time and constant-program-size compilation with respect to the number of finite volumes, together with up to two orders of magnitude speedup in simulation time when compared with the most prominent open-source competitor.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Modeling and simulation*.

Additional Key Words and Phrases: MLIR, LLVM, modelica

ACM Reference Format:

Michele Scuttari, Daniele Cattaneo, Federico Terraneo, Giovanni Agosta, and Stefano Cherubin. 2026. MLIR Dialects for DAE-based Modeling Languages. *ACM Trans. Arch. Code Optim.* 23, 3, Article 109 (September 2026), 25 pages. <https://doi.org/10.1145/3836762>

1 Introduction

A large number of engineering domains [42, 46, 53], 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, 59] are becoming *smart* to account for the variability of renewable energy sources and the increased demand introduced by

This work was supported by the Italian Ministry of University and Research (MIUR) under the National Recovery and Resilience Plan (PNRR), by the European Union (EU) under the NextGenerationEU project, by LWT³, and by Huawei Italy Research Center. The source code of MARCO is freely available on GitHub <https://github.com/marco-compiler/marco>.

Authors' Contact Information: Michele Scuttari (corresponding author), DEIB, Politecnico di Milano, Milano, Italy; e-mail: michele.scuttari@polimi.it; Daniele Cattaneo, DEIB, Politecnico di Milano, Milano, Italy; e-mail: daniele.cattaneo@polimi.it; Federico Terraneo, DEIB, Politecnico di Milano, Milano, Italy; e-mail: federico.terraneo@polimi.it; Giovanni Agosta, DEIB, Politecnico di Milano, Milano, Italy; e-mail: giovanni.agosta@polimi.it; Stefano Cherubin, IDI, Norwegian University of Science and Technology, Trondheim, Norway; e-mail: stefano.cherubin@ntnu.no.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1544-3973/2026/09-ART109

<https://doi.org/10.1145/3836762>

electric vehicles. Similarly, industrial production is being revolutionized by Industry 4.0 technologies including digital twins [15, 38, 39], 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.

Focusing on simulation models based on Differential-Algebraic Equations (DAEs), dedicated modeling languages exist to express the underlying physics of a system in a computational form. Such modeling languages offer constructs that, when combined, define a system of differential and algebraic equations, 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. It follows that 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.

Compiling a declarative representation of a DAE system to the imperative instruction set of computer architectures introduces representation issues, which the compiler's internal Intermediate Representation (IR) should address. The main difficulty is in designing a compiler that can preserve the declarative nature of the model in the initial compilation steps – whose algorithms come from the mathematical domain – as well as efficiently lower such representation in a procedural form for the later compilation steps. Recent developments in mainstream compiler technology – such as the **Multi-Level Intermediate Representation (MLIR)** project [37] – easily adapt to this scenario by providing developers the capability of defining custom layers of IRs with semantics that best fit each aspect of the compiled language.

1.1 Contribution

This article advances the state-of-the-art in DAE-based compilers with the following contributions.

- We present the first – to the best of our knowledge – MLIR dialect targeting DAE-based modeling languages. More specifically, our work uses Base Modelica as reference source language, although the concepts involved have a more general applicability. The proposed design preserves the declarative nature of Base Modelica models while enabling modular and extensible IRs that support state-of-the-art analyses and transformations specific to DAE manipulation, such as the array-aware matching algorithms described in [24] and [40].
- We demonstrate how this dialect can be implemented in the prototype compiler **MARCO (Modelica Advanced Research Compiler)** [9] to achieve a full MLIR-to-LLVM compilation pipeline. In doing so, we introduce additional MLIR dialects to integrate with the IDA solver [32] and the runtime environment of MARCO.
- We compare this proof-of-concept implementation against a state-of-the-art compiler solution, and demonstrate significant gains both on compilation time and simulation execution time, trending towards manually optimized C++ reference implementation.

1.2 Article Structure

The remainder of the article is organized as follows. In Section 2, we describe the main characteristics of DAE-based modeling languages, we explain the transformations that need to take place within a compiler when dealing with such representations, and we highlight the related challenges. In Section 3, we introduce some background concepts and notation on the multi-layered approach we adopted in our solution. In Section 4, we compare our compiler design to other existing solutions in this domain, and in the compiler field more broadly. In Section 5, we present the MLIR dialects that have been implemented to address the typical problems of DAE-based modeling languages

and achieve the compilation of a Base Modelica model. In Section 6, we evaluate the effectiveness of our work by analyzing the impact of our MLIR-based design within the compilation pipeline. We compare code sizes, compilation times, and simulation times of our compiler with respect to an industry-grade solution. Finally, in Section 7, we draw some conclusions and discuss future works.

2 Background

In this section, we introduce the basic concepts of DAE-based modeling languages that are relevant to our work, and we discuss their unique characteristics via comparative examples with the most common programming languages. From there, we present the key mathematical aspects of DAEs, as well as the steps required for their numerical integration. These steps imply both the mathematical transformations bringing the DAE system into a computable form, and the generation of executable code.

2.1 DAE-Based Modeling and Modeling Languages

To understand the purpose of DAE-based modeling languages, we first have to introduce the concept of a *model* in engineering. In this context, a model is a set of equations that describes the relations between the relevant physical quantities of an object or a system. Said quantities could be voltage and current in a power line, rotational speed and torque in a motor, temperature and reaction rate in a chemical reaction, and many more examples could be given in these as well as in other engineering domains. Due to the nature of the physical laws underpinning engineering domains, said equations are in the general case differential. Performing a simulation amounts to numerically integrating over time the equations composing the model, and obtaining the evolution in time of the physical quantities of interest.

The methods and tools used to perform simulations largely depend on the nature of the differential equations involved. A significant distinction is the one between Partial Differential Equations (PDEs) and Differential-Algebraic Equations (DAEs). The former are often addressed by specialized software packages due to the variety in their mathematical formulation, spatial domains, and boundary conditions. **Computational Fluid Dynamics (CFD)** software is an example of such an application. The latter is a versatile class of models that proved capable of lumped parameter modeling in a large number of domains and nonetheless represents the outcome of spatial discretization methods that are widely used in the numerical approximation of PDE solutions.

Various DAE-based modeling languages have been introduced, such as Omola [12], gPROMS [16], Simscape [43], and Modelica [44]. Since the numerical integration of engineering-scale models is a computationally intensive task, modeling languages are most often compiled rather than interpreted. Historically, DAE-based languages are compiled using a two-step process: the source model is first processed by a *model translator* that performs the compilation steps specific to the DAE domain using custom-built compiler infrastructures developed independently by each vendor; then, the translator produces procedural simulation code in a language such as C, which is compiled using an ordinary C compiler. Differently from general-purpose programming languages, the produced simulation executable program is usually run *only once* to produce the simulation output — that is, the evolution of the modeled quantities in time. Due to this characteristic, the compilation time and the numerical integration time both contribute to the time-to-solution metric, and research efforts in their individual improvement are equally important.

2.2 The Modelica and Base Modelica languages

Among the DAE-based languages, Modelica gained considerable traction thanks to its open and community-based language specification, which is implemented both by open-source and

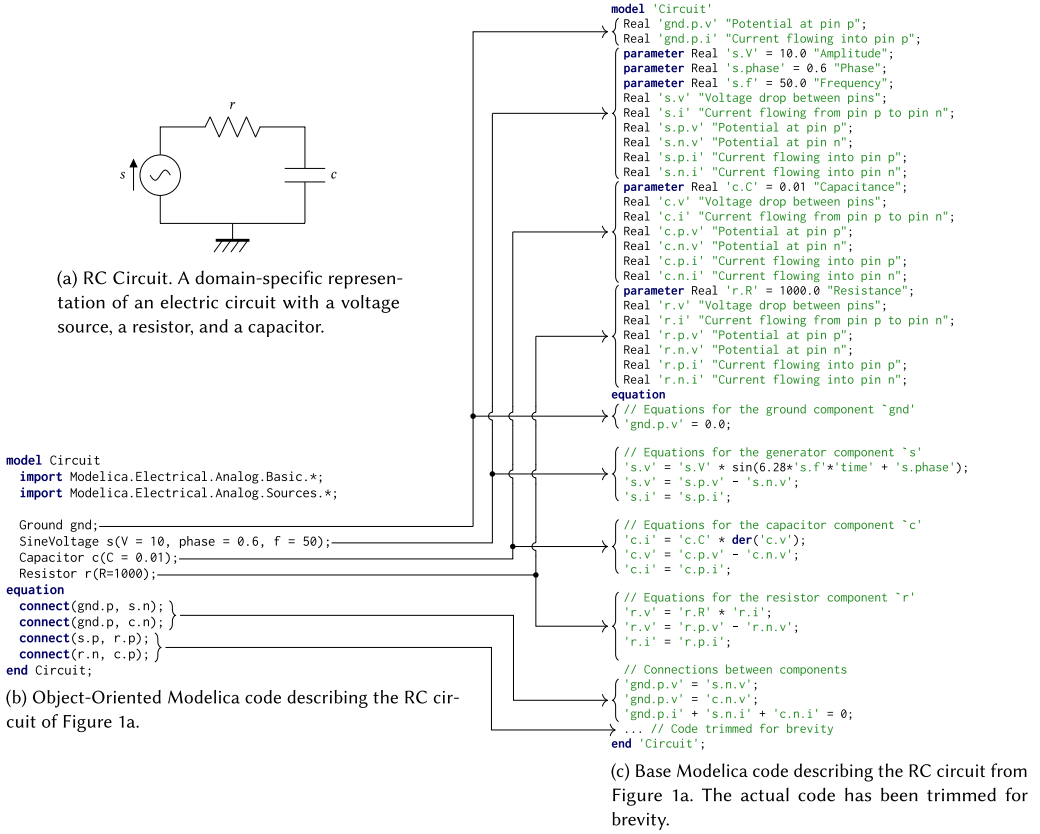


Fig. 1. Correspondence between the Object-Oriented Modelica code describing an RC circuit and its Base Modelica representation. It should be noted that each statement in the Modelica code corresponds to one or more statements in the Base Modelica code.

commercial environments. For this reason, from now on we will discuss the concepts of DAE-based languages using the Modelica syntax, although the presented concepts are applicable to other modeling languages as well.

Modelica is an object-oriented, declarative, multi-domain modeling language. To support the modeling of complex systems, Modelica supports the hierarchical composition of models starting from simpler ones. For example, an electrical circuit such as the one in Figure 1(a) can be modeled by instantiating and connecting models of its components such as resistors and capacitors, as shown in Listing 1(b). Modelica is an object-oriented language, thus it is possible for a model to extend other models and implement interfaces. For example, both the Resistor and Capacitor models in the Modelica standard library derive from the *OnePort* interface – exposing an abstract electrical component with two pins *p* and *n* – and only need to add their characteristic equation, respectively $v = r * i$ and $i = c * \text{der}(v)$, with the *der* operator representing the derivative over time.

The key construct in the Modelica syntax is the *model*. For the purpose of this article, we can assume models to be composed of only two blocks, the first for declaring variables, parameters and constants, and the second for listing the model equations. Modelica is a declarative language,

and while equation statements – such as the ones just shown for the resistor and capacitor – resemble assignment statements in programming languages, they are not. In Modelica, the `=` operator represents an equality in the mathematical sense and not an assignment. For example, Ohm's law in the Resistor model could just as well be written as $0 = v - r * i$ and that would be valid Modelica code even though, clearly, it is not a valid assignment. One more consequence of the language being equation-based is that the order in which the equations are written does not matter. Statements in a model's equation block can be arbitrarily shuffled without altering the model behavior. Additionally, the number of equations in a model must always be equal to the number of variables. Otherwise, the underlying system of equations is either under- or over-determined. The model in Listing 1(b) also makes use of the `connect` statement, which the language provides to automatically generate the equations required for connecting model components (e.g., Kirchhoff's laws in electrical circuits).

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 [35], is currently in the process of being standardized. Tools already exist to convert Modelica code to Base Modelica – a process called *flattening* – thus making it a viable source language to research advanced compilation techniques for DAE-based languages. Indeed, Base Modelica allows to focus 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 [3].

The circuit of Listing 1(b), when flattened into Base Modelica code, reads as Listing 1(c) (with some trimming for space reasons). 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.

Optimization of algorithms and data structures can be especially impactful when leveraging the concept of *equation arrays*. Equation arrays model the concept of a set of equations, parametric to an index, such as

$$x_i(t) = 1 - \dot{x}_i, \forall i \in [1, 10]$$

The above equation can be expressed in Base Modelica as

```
for i in 1:10 loop
  x[i] = 1 - der(x[i]);
end for;
```

Although this representation may appear simple, it represents a relevant source improvement. Indeed, in the state-of-the-art of DAE systems simulations as of Kofman et al. [20], it was deemed necessary to dismantle this construct into a series of ten equations over the same number of scalar variables. In current industry-grade compilers, this task is applied as soon as possible [8], and the internal compiler IR has no capability to represent `for` constructs at all. However, since then, it has been found that it is possible to preserve `for` constructs in many significant cases, with associated code size and performance improvements [24]. Therefore, in a modern DAE-based compiler, preserving `for` constructs in multiple stages of the IR is of great importance.

2.3 An Overview of Differential-Algebraic Equations

In order to support the modeling of complex systems in an expressive and maintainable way, component-based modeling is a necessity. However, the characteristic equations of individual models are in the general case differential, while the equations arising from `connect` statements are algebraic. Thus, when the equation statements are combined, they eventually lead to a system of DAEs.

In mathematical notation, a DAE system appears as in Equation (1), in which $\mathbf{d}$ is a vector of n differential variables, $\dot{\mathbf{d}}$ is a vector of derivative variables such that $\dot{\mathbf{d}}_i(t) = \frac{d\mathbf{d}_i(t)}{dt}$, $\mathbf{a}$ is a vector of m algebraic variables, t is the independent variable (usually time), and F is a vector of $n + m$ functions involving time, the variables, and their derivatives.

$$F(\mathbf{d}(t), \dot{\mathbf{d}}(t), \mathbf{a}(t), t) = 0 \quad (1)$$

Numerically solving DAE systems is a complex mathematical topic, which requires to first introduce the concept of (*differential*) *index*. For a given DAE system, its index is the minimum number of differentiations that need to be applied to the algebraic equations to obtain a system of explicit Ordinary Differential Equations (ODEs) in the form of Equation (2), where $\mathbf{x}$ is the vector of model state variables, and $\mathbf{y}$ is the vector of output variables.

$$\begin{cases} \dot{\mathbf{x}} = f(\mathbf{x}, t) \\ \mathbf{y} = g(\mathbf{x}, t) \end{cases} \quad (2)$$

An explicit ODE is therefore an index-0 DAE. To be so, it must not contain algebraic loops or structural singularities. An index-1 DAE system contains algebraic loops but no structural singularities [20]. DAE systems with higher index are structurally singular and – if not considering algorithms working only in very specific cases – need to be rewritten in an index-1 form to be solved. It should be noted that some models arising from relevant engineering domains, when written in their most natural form, do result in higher-index DAE systems. However, there are symbolic manipulation techniques that can be used to rewrite them in index-1 form in an automated way, an operation called *index reduction* [45]. An example of index reduction is the Pantelides algorithm [16]. Once the system of DAEs is in index-1 form, broadly speaking, it can be solved in one of two possible ways, each of which has its unique tradeoffs that make them preferable in different contexts.

The first method is to choose to use a DAE solver algorithm to perform the numerical integration. Numerical solvers exist – in the form of software libraries such as IDA in the SUNDIALS suite [32] – to directly perform the numerical integration of index-1 DAE systems. Such software libraries require the compiler to provide functions (in the computer science meaning) to perform computations on the DAE system, such as computing the value of the residuals vector and of the Jacobian matrix for a given candidate solution of the DAE system. When following this first method, the compiler's task is to generate efficient code to perform those computations [8], and produce an executable program that links with the appropriate solver libraries. The model simulation performance however also greatly benefits from optimizations performed through symbolic manipulation of the model, as passing each and every equation directly to a DAE solver can be inefficient [55].

The second method is instead to proceed with further symbolic manipulation steps with the aim of obtaining an explicit ODE system, and then performing numerical integration using an ODE solver algorithm. The symbolic manipulation required is a multi-step process that is collectively called *causalization*. It involves, at minimum, the *matching* and *scheduling* steps [24]. The equation-variable matching step pairs each variable (and derivative) to an equation that is used to compute its value. The scheduling step attempts to find an order between the equations so that no variable is used before it is computed, allowing the equations to be lowered into assignment statements. In cases where this is not possible, the scheduling step evidences the presence of algebraic loops that need to be addressed through further symbolic manipulation techniques such as *tearing* [20].

Once the system is in explicit ODE form, a wide range of numerical solver algorithms can be used, each of them differing in characteristics such as time-to-solution and numerical stability. These characteristics make them suitable for different classes of problems. Summarizing for brevity,

these algorithms work iteratively by first computing the derivatives of the state variables, and then using the derivatives to compute the value of the state variables at the next timestep. The simulation timestep – depending on the algorithm – can either be a fixed value or can be dynamically adapted to keep the simulation error within desired bounds.

When performing the causalization of the DAE system, the compiler needs to perform the entire symbolic manipulation pipeline, generate efficient lowered code, and connect it with the ODE solver code. It should be noted that for some solver algorithms, the compiler can generate the entire numerical integration code without linking with external solver libraries [7, 32]. An example is when using the *inline integration* technique [21].

Choosing between DAE and ODE solvers is a model-dependent decision, and involves multiple criteria whose analysis goes beyond the scope of this article. For example, DAE solvers can reduce the number of Newton iterations needed to solve algebraic loops, but they may behave differently – from a numerical point of view – in case of small sizes of the numeric integration step. Both methods require proper initialization to begin numerical integration, a task whose complexity can range from trivial to very complex on a model-specific basis. This task requires to solve a non-linear system of (non-differential) equations and the compiler must therefore be capable of linking the generated code with the external nonlinear solvers [32].

In this article, we analyze the implications of direct DAE solving and DAE causalization in terms of compiler development. The reader that is interested in their mathematical foundations is referred to the existing literature [20].

3 MLIR as an IR for DAE-Based Languages

From the previous section, it emerges that DAE-based *modeling* languages generate a diverse and unique set of challenges compared to all *programming* languages, regardless of their paradigm. These challenges arise from the compiler having to deal both with the mathematical and computer domains, needing to manipulate both equations and variables, and – later in the process – procedural code. It follows that operations involving the two classes of representations require very different data structures. On one hand, the manipulation of procedural code can be trivially mapped onto well-known compiler intermediate representations used in general-purpose programming languages and tools [36]. On the other hand, the DAE-based transformations do not find an obvious mapping onto existing data structures within the mainstream compiler tools.

Recent developments in compiler technology – such as the MLIR project – provide a software infrastructure that easily adapts to scenarios such as this one, where clearly different and well-separated abstraction levels manifest the need for a MLIR.

Although the approach to code compilation based on multiple intermediate representations is well-known in the state of the art, and industrial compilers, such as gcc, have been built on this principle decades ago, the novelty brought by the MLIR framework consists of the interoperability of its composing elements, based on a common syntax. By leveraging the multiple levels of abstraction in MLIR, the code transformations become easier to implement and more effective in enabling optimizations – especially when every domain-specific concept is mapped onto a separate level of abstraction. To familiarize the reader with the MLIR framework, the remaining part of this section is devoted to the description of the MLIR syntax, semantics, and capabilities. However, given the breadth of the topic, such treatment is limited to give the information needed for a proper understanding of this article. The reader interested in a deeper knowledge of the framework is referred to the official documentation and tutorials [1, 2].

In MLIR, each abstraction level is called *dialect*, and its design is based on concepts such as *operations*, *types*, and *attributes*. A dialect typically represents the semantics specific to a certain domain (e.g., linear algebra, memory management), while the framework itself remains completely

independent from these semantics. New dialects can be defined to construct a compiler for any language, and they can also be intermixed with already existing ones to reuse their semantics and optimizations. Their interoperability is empowered by interfaces, which enable transformation passes to operate on the IR based solely on relevant properties, without requiring knowledge of the entire operation semantics. The modularity of dialects and their interoperability is also complemented by the progressive lowering of the employed abstractions, with the objective of avoiding the premature loss of higher-level semantics and metadata. This flexibility enables the use of MLIR within an exceedingly wide variety of use cases, from conventional imperative programming languages to graph-based languages such as the ones employed for high-level synthesis [10]. In our work, to the best of our knowledge, we provide the first MLIR dialects for DAE-based modeling languages.

MLIR is part of the greater LLVM project [36], which is an industry-grade compiler construction framework. As such, MLIR has been designed for high compatibility and manifests multiple similarities with its umbrella project, especially to allow the reuse of the LLVM-IR and the LLVM backend. Despite the similarities, an important difference between MLIR *operations* and LLVM-IR *instructions* lies in the possibility of the former to be nested. Indeed, operations can be enriched with one or more *regions* that may contain multiple code blocks. Like LLVM-IR, MLIR employs a Static Single Assignment (SSA) representation, where a *value* can be only statically defined once within its scope, and hence a more specific term we use is *SSA value*. It is also important to note that operations may or may not produce a result. If a result is produced, that is stored in a *value* whose life is restricted to the region in which it is defined. In the MLIR code examples, *values* are denoted with the “%” prefix. Additionally, MLIR operations provide an alternative to *values* in the form of the *symbol* concept, which is an alias (or reference) for an operation – with or without *value*. Subsequent operations may take a symbol as an *attribute* to represent an access to that symbol. In the MLIR code examples, symbols will be prefixed with “@”.

4 Related Works

The MLIR project and – more broadly – its layered approach are becoming more and more popular in the field of compiler development due to the flexibility and expressiveness they provide. Applications areas cover very different domains, from machine learning [4, 5, 34] to high-level synthesis [23, 60].

The abstraction capability of MLIR has also been used to improve the compilation of well-established languages. For example, the Polygeist project applies polyhedral optimization techniques to the C language [47]. Ongoing efforts aim for a ground-up re-implementation of one of the most popular Fortran compilers – Flang – leveraging the MLIR infrastructure. The new performance-oriented structure of Flang provides useful insights for our design. An MLIR dialect called **Fortran IR (FIR)** [37] has been designed to capture the semantics of the multiple Fortran language versions and act as a common denominator to exploit their features. The current design of Flang aims at lowering FIR to LLVM-IR via the LLVM dialect provided by MLIR, thus reusing most of the existing middle-end and back-end of LLVM. Core dialects of the MLIR framework are selectively reused and exploited as needed within the project, e.g., to represent functions – *func* dialect – and to express OpenMP parallelism concepts – *omp* dialect. Integration with additional dialects has been demonstrated as feasible and effective, especially for improving the performance of specific use cases [17].

The MLIR ecosystem and its layered approach have been also used in the context of programming reactive systems, bridging the gap between concepts and algorithms from synchronous languages and the abstractions developed within general-purpose compilers [50]. However, despite the shared concept of *equation* – in the sense of equality between two expressions – such languages do not support DAEs nor a-causal modeling semantics; this characteristic narrows their approach to a less

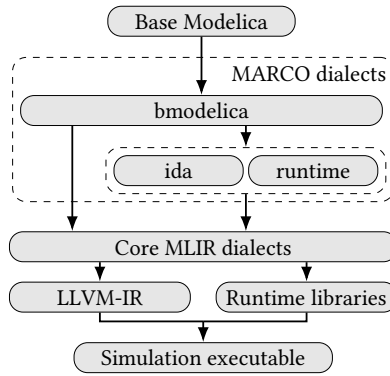


Fig. 2. Overview of the MARCO compilation pipeline: from Modelica source to simulation executable, highlighting the intermediate representations, custom MLIR dialects, and integration with external solvers and runtime libraries.

general kind of mathematical problem than that of DAE-based languages such as Modelica [49]. Thus, they are optimized and better suited for very different application domains, such as designing graphical user interfaces through a declarative approach [14].

As for the Modelica language, other compilers exist, but none of them take advantage of multi-level representations. Dymola [22] is a proprietary solution developed by Dassault Systèmes. Being a closed-source solution, no knowledge about its internal structure is publicly available. JModelica [11] is a discontinued open-source solution whose aim was to bridge the gap between the need for high-level description languages and the details of numerical optimization algorithms. The JModelica compiler supported a subset of the Modelica language and was developed using the JastAdd framework [31]. The **OpenModelica Compiler (OMC)** [26] is an open-source solution developed by the Open Source Modelica Association and represents the most popular open-source solution within the community. It is written in MetaModelica [27, 28, 51] – a super-set of the Modelica language – and its frontend can emit the Base Modelica code for each given Modelica project provided as input, thus enabling interoperability with other tools such as MARCO. Within the context of large-scale modeling, ongoing efforts to improve DAE-based compilers focus on the preservation of arrays [24, 52] – or their reconstruction [6, 54] – throughout the compilation pipeline. A layered approach – such as the one implemented by MLIR – represents a structured and reliable opportunity to achieve this goal without loss of semantics along the process.

5 MLIR Dialects for DAE Systems

In this section, we introduce the three MLIR dialects that have been designed to perform the compilation of a Base Modelica model within the open-source MARCO compiler, whose pipeline is illustrated in Figure 2. Following the multi-layered approach of MLIR, these dialects aim to preserve the high-level concepts until the necessary transformations are applied while moving towards the low-level LLVM-IR code, thus enabling more optimization opportunities throughout the compilation pipeline. More in detail, we are particularly interested in introducing these dialects to enable the explicit representation of array variables and array equations, whose semantics are explained in Section 2.2.

The `bmodelica` and `ida` dialect reflect the concepts explained in Section 2.3. The former maps the source language into the MLIR ecosystem and offers the necessary capabilities for optionally going through the causalization process. The latter is used when interfacing with the IDA software

library, which is used to showcase the usage of a DAE solver. Finally, the `runtime` dialect captures the information and procedures that are necessary to the runtime environment to operate correctly.

5.1 Base Modelica Dialect

From the compiler's perspective, DAE-based modeling languages describe a system by defining a set of equations that govern the evolution of variables over time. To accurately capture this modeling intent, the IR should distinguish between the system of equations as a whole, individual DAEs, variables, and the time being the independent variable of the system.

In this section we describe the `bmodelica` dialect, which is used to represent these entities within the MLIR ecosystem. We examine the difficulties of mapping the high-level semantics of DAE systems onto an IR of a compiler, and analyze which constructs of the MLIR domain can be usefully employed to overcome this challenge.

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. In MLIR, this property is well captured by the `graph region` construct, which does not pose constraints on the order of the operations contained within. 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. Such a block belongs to the unique region of the operation, characterized by the aforementioned graph semantics.

Variables as Symbols vs SSA Values. 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. During the design of the IR, two alternatives were considered for the semantics of this operation: SSA values or symbol semantics. The latter have been preferred to avoid scoping constraints within the IR. Indeed, MLIR operations may be characterized by the *isolated-from-above* trait, which prohibits, within their regions, to reference SSA values defined in parent scopes. This restriction would complicate variable referencing when using SSA value declarations, whose choice for the operation's semantics would therefore potentially reduce the interoperability of the operation with other MLIR dialects. As a result, the `variable` operation does not produce SSA values, and the symbol attribute indicating the variable name is also used to reference the variable. Referencing is performed through the `variable.get` operation, which will be later introduced in this section to shift from symbol to SSA semantics within the body of equations.

A Dedicated Type System. 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 array of elements, and can be omitted in the case of scalar variables. The `parameter` and `constant` properties are of course 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`), 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. The introduction of these basic types, as opposed to the reuse of built-in ones, allows to decouple the source language representation from the machine-level specificities. By avoiding a fixed bitwidth, the decision on the actual precision can be deferred to later stages of the compilation pipeline, where it can be determined based on model-specific properties and numerical requirements. In the

examples provided within this article, we stick with built-in types to keep the textual representation as compact as possible, but nonetheless can be replaced with the Base Modelica equivalents.

Capturing the Generic Structure of 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, respectively decoupling the generic structure of an equation from the optional indices it iterates on. An `equation_template` operation has a single block containing the syntactic tree of the equation. The root of this 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 of course 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, with SSA values defined exclusively within the block of the equation template. Multiple expressions may also share the same SSA values, enabling efficient reuse within the block. As anticipated, the referencing to variables is performed through the `variable_get` operation. 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 semantics of this operation also do not enforce a single transition from symbol to SSA value, meaning that multiple `variable_get` operations can reference the same variable within a single equation. The time variable – which is always implicitly declared within a model and constitutes the independent variable for DAE systems – can be referenced by 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, these 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 the `equation_instance` operation. This operation takes a reference to the identifier – expressed again in the form of SSA value – of an equation template. 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. The two sets – of initial and dynamic equations – represent two different systems of equations that are solved independently: the system of initial equations, valid only during initialization, is entirely algebraic and solved once at the beginning of the simulation to determine the initial values of the variables; the system of dynamic equations, valid for the whole duration of the simulation, is instead a DAE system whose variables are updated repeatedly throughout the simulation as the virtual time progresses. In both cases, however, the possible placement of `equation_instance` operations is limited to the respective blocks of these two operations (or at most to other non-isolated `bmodelica` operations such as the `scc` one, which is introduced later in this section).

As previously mentioned, the use of SSA semantics for variables is not feasible due to the flexible scoping required for variable references. In contrast, equations have restricted placement within the IR – as described by the proposed dialect – making SSA values suitable for their representation. While symbol semantics could technically be applied also to equations, SSA semantics are preferred because they simplify the implementation of IR transformations: MLIR automatically manages SSA

values in case of modifications, whereas symbol tables require manual and potentially error-prone handling. Consequently, adopting SSA semantics for equations is both practical and advantageous.

The `equation_instance` is also optionally enriched by an `indices` attribute providing the multidimensional range of indices of the equation. Compared to upstream MLIR dialects, this attribute can be seen as a restricted form of the `scf.for` operation, allowing only constant ranges and steps equal to one. Even though this choice may seem limiting at first, it should be noted that the iteration space of array equations in Base Modelica must always be compile-time computable, as structurally relevant for the system of DAEs (i.e., it is needed for the mathematical analyses such as matching and scheduling), and non-unitary steps can still be obtained by modifying the accesses to variables within the equation body.

An example. Figure 3 shows an example of a Base Modelica model (Listing 1) and its corresponding representation in the `bmodelica` dialect (Listing 2). The code represents a simple 1D thermal model, which describes thermal conduction in a solid copper wire divided into 1000 sections of equal length. 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. The variable named `τ` 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. The assignments to parameters are treated as initial equations, as being performed only once during the initialization phase. 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 `τ` are referenced within the equation bodies by using the `variable_get` operation. The variable `τ`, being an array variable, is obtained in the form of a tensor having multiple elements. Its individual elements are then obtained by leveraging the `tensor` dialect. The `arith` dialect, also already part of the MLIR codebase, is also used while representing the two sides of the equation. 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 MLIR are subject to an offset of -1. 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 `τ`. The `time` operation does not appear in the MLIR listing for this example, given that the time variable is indeed not used in the original Base Modelica code. The reader interested in its usage is referred to the IR printouts that are made available for the heat exchangers model considered in Section 6, in which the time variable is referenced within the equations.

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 system state over time. However, this mathematical construct is not directly understandable by a machine and must be materialized into more practical, machine-friendly, representations. In the most general case, the value of a derivative must be stored somewhere in memory, so that later computations can fetch and use it while updating other variables. Such memory location is reserved by declaring additional variables, one for each variable whose derivative is taken in the source model. The derivative operations are then replaced with references to these newly inserted variables. A specifically designed attribute is also attached to the model to inform later stages within the compiler about which variables have been inserted during this transformation. This attribute – named `der` – consists of a list of tuples (a, b, i) , in which a and b are symbols and i is an

model Thermal1D

parameter Real g = 0.00314785;
parameter Real c = 0.2707936;
Real T[1000];

equation

```
for i in 2 : 999 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[1000]) = g * (T[999] - T[1000]) +
                  2*g * (293.15 - T[1000]);
```

initial equation

```
for i in 1 : 1000 loop
  T[i] = 293.15;
end for;
```

end Thermal1D;

Listing (1) 1D thermal model of a solid copper wire described using the Base Modelica language.

```
bmodelica.model @Thermal1D {
  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 = arith.constant 1 : index
    %1 = arith.constant 2 : index
    %2 = bmodelica.variable_get @c : f64
    %3 = bmodelica.variable_get @g : f64
    %4 = bmodelica.variable_get @T : tensor<1000xf64>
    %5 = arith.subi %1, %0 : index
    %6 = tensor.extract %4[%5] : tensor<1000xf64>
    %7 = bmodelica.der %6 : f64 -> f64
    %8 = arith.mul %2, %7 : f64
    %9 = arith.subi %1, %1 index
    %10 = tensor.extract %4[%9] : tensor<1000xf64>
    %11 = arith.sub %10, %6 : f64
    %12 = arith.mul %3, %11 : f64
    %13 = tensor.extract %4[%i] : tensor<1000xf64>
    %14 = arith.sub %13, %6 : f64
    %15 = arith.mul %3, %14 : f64
    %16 = arith.add %12, %15 : 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 = arith.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 (2) 1D thermal model of a solid copper wire described using the **bmodelica** dialect. This code also reuses the **arith** and **tensor** core dialects of MLIR.

Fig. 3. Correspondence between Base Modelica code describing a the 1D thermal model of a copper wire and its representation in the MLIR Base Modelica and core dialects. A summary of the relevant **bmodelica** operations is provided in Table 1.

optional list 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). Following the example of Listing 2, the result of the derivatives materialization process is shown in Listing 3. A new variable named `der_T`, of the same type of T , has been declared, and the `der` attribute has been attached to the model operation to remember the relationship between the two variables. The `der` operation is then replaced within the equation bodies with references to the new `der_T` variable. Only the first equation body has been reported, but the process also applies to the remaining ones.

Table 1. Summary of the `bmodelica` Operations Used in Listing 2 to Represent a Base Modelica Model in the MLIR Domain

Operation	Purpose
<code>model</code>	Declares a Base Modelica model, encapsulating variables and equations within its unique region.
<code>variable</code>	Declares a variable within the model, specifying its name, type, and optional attributes (e.g., parameter, constant, dimensions).
<code>initial</code>	Contains the initial equations, which are solved once at the start of the simulation.
<code>dynamic</code>	Contains the dynamic equations, which are repeatedly computed during the simulation at each time step.
<code>equation_template</code>	Defines the generic structure of an equation, optionally parameterized by induction variables for array equations.
<code>equation_instance</code>	Instantiates an equation (or array of equations) from a template, specifying the actual indices and initial / dynamic nature.

```

bmodelica.model @Thermal1D attributes {
  der = [#bmodelica.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 inductions = [%i] {
    %0 = arith.constant 1 : index
    %1 = arith.constant 2 : index
    %2 = bmodelica.variable_get @c : f64
    %3 = bmodelica.variable_get @g : f64
    %4 = bmodelica.variable_get @T : tensor<1000xf64>
    %5 = arith.subi %i, %0 : index
    %6 = tensor.extract %4[%5] : tensor<1000xf64>

    %der_T = bmodelica.variable_get @der_T : tensor<1000xf64>
    %der_T_i = tensor.extract %der_T[%5] : tensor<1000xf64>

    %8 = arith.mulf %2, %der_T_i : f64
    %9 = arith.subi %1, %1 index
    %10 = tensor.extract %4[%9] : tensor<1000xf64>
    %11 = arith.subf %10, %6 : f64
    %12 = arith.mulf %3, %11 : f64
    %13 = tensor.extract %4[%i] : tensor<1000xf64>
    %14 = arith.subf %13, %6 : f64
    %15 = arith.mulf %3, %14 : f64
    %16 = arith.addf %12, %15 : f64
    %lhs = bmodelica.equation_side %8 : tuple<f64>
    %rhs = bmodelica.equation_side %16 : tuple<f64>
    bmodelica.equation_sides %lhs, %rhs : tuple<f64>, tuple<f64>
  }

  bmodelica.dynamic {
    bmodelica.equation_instance %t0, indices = {[2,999]}
  }
  // Omitted for brevity.
  ...
}

```

Listing 3. MLIR representation of the model of from Listing 2 after the materialization of derivatives. Only relevant aspects of the resulting IR – i.e., the derivative variable declaration, its usages, and the attribute placed on the `model` operation – are shown for brevity.

Causalization. Different algorithms can be employed in this process, with the aim of reducing the computational effort [24, 25]. Although the specifics of the algorithms employed are beyond the scope of this article, we should analyze how the results of the matching and scheduling tasks reflect on the IR. During the matching phase, each `equation_instance` operation is annotated with a `match` attribute. This attribute explicitly identifies which variable is matched with the equation, using the variable’s symbol (name) and, if the variable is an array, the specific indices of the matched scalar elements. The data dependencies between equations, made explicit by the results of the matching phase, are then analyzed. The Strongly Connected Components (SCCs) are identified within the dependency graph, and represented in the form of `scc` operations, which contain one or more `equation_instance` operations. Equations that are not part of cyclic dependencies are then enriched with an additional attribute named `schedule`, consisting of a list of *iteration directions* to be followed while visiting each scalar equation during the execution of the simulation. One direction is given for each induction variable: a forward direction implies an increment of the induction variable, while a backward direction implies its decrement. The SCCs are inserted inside a `dynamic` or `initial` operation, according to the nature of the equations the compiler is dealing with, and a `schedule` operation is used as an external wrapper. This `schedule` operation can be referenced through its symbol attribute, and the execution of its SCCs can be invoked from elsewhere within the IR by using the `run_schedule` operation. It should be noted that, during both the matching and scheduling phases, the indices of equation arrays may be split if the solutions are not found to be invariant across the whole iteration space. This split greatly benefits from the decoupling between the equation bodies – generic in their indices – and the actual instances. Indeed, the insertion of multiple `equation_instance` operations, or their enriched versions, is a quick and memory-light task, which reuses the already existing `equation_template` operations and therefore does not need to clone all the equation expressions. The intermediate representations used for the causalization process are exemplified in Listings 4 and 5,

Table 2. Summary of the Causalization-Related Constructs in the `bmodelica` Dialect

Construct	Kind	Purpose
<code>match</code>	Attribute	Specifies which variable (and indices) an equation instance is matched to.
<code>schedule</code>	Attribute	Indicates the iteration directions for induction variables, guiding execution order to respect data dependencies.
<code>scc</code>	Operation	Groups equation instances that form a strongly connected component in the dependency graph, representing cyclic dependencies.
<code>schedule</code>	Operation	Encapsulates a scheduled set of SCCs, defining their execution order and enabling invocation from other IR locations.
<code>run_schedule</code>	Operation	Invokes the execution of a <code>schedule</code> operation.

Notice that, despite the shared name, the `schedule` attribute and the `schedule` operation have a different nature and purpose.

```

bmodelica.model @Thermal1D attributes {
  der = [#bmodelica.der<@T, @der_T, <[0,999]>>]
} {
  ...

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

  bmodelica.dynamic {
    bmodelica.scc {
      // Matched with der_T[i] (in MLIR, @der_T[i - 1])
      bmodelica.equation_instance %t0, indices = {[2,999]},
      match = <@der_T, {[2,999]}>
    }
  }
}
...
}

```

Listing 4. Code fragment that represents the model from Listing 3 after the matching stage.

```

bmodelica.model @Thermal1D attributes {
  der = [#bmodelica.der<@T, @der_T, <[0,999]>>]
} {
  ...

  bmodelica.schedule @schedule_0 {
    bmodelica.dynamic {
      bmodelica.scc {
        bmodelica.equation_instance %t0, indices = {[2,999]},
        match = <@der_T, {[2,999]}>,
        schedule = [forward]
      }
    }
  }
  ...
}

func.func @runEquations() {
  bmodelica.run_schedule @Thermal1D::schedule_0
  func.return
}

```

Listing 5. Code fragment that represents the model from Listing 4 after the scheduling stage.

which show a representative set of modifications respectively performed by the matching and scheduling stages. A summary of the causalization-related constructs in the `bmodelica` dialect is also provided in Table 2.

In Listing 4, the reported `equation_instance` operation is enriched with the `match` attribute, specifying that the indices 2 to 999 of the equation are matched with the same range of indices of variable `der_T`. In the example of Listing 5, the equation is enriched with an additional `schedule` attribute which specifies the iteration directions that need to be followed to respect data dependencies while computing the value of each scalar variable. In this case, the iterations are performed by increasing the index value from 2 to 999, as specified by the `forward` keyword. The complementary backward would have been chosen if the iteration were to be performed in the opposite direction, from 999 to 2. Finally, the execution of the schedule is exemplified inside the `runEquations` function. The name and the body of this function are completely arbitrary, and aim only to demonstrate the decoupling between the schedule itself, and the points within the IR where it is invoked. In real-world applications, the `runEquations` function may have a more complex logic.

5.2 The *ida* Dialect

As anticipated in Section 2, an alternative to causalization is the direct usage of a DAE solver. MARCO supports – through a dedicated dialect – the **Backward Differentiation Formula (BDF)** integration method [20] implemented by the SUNDIALS IDA solver [32]. The **Application Programming Interface (API)** of the solver is wrapped in a dedicated adapter – provided by

Table 3. Summary of the Main Operations of the `ida` Dialect

Operation	Purpose
<code>instance</code>	Declares an instance of the IDA solver.
<code>init</code>	Initializes a solver instance, allocating memory and preparing it for use.
<code>getter</code>	Declares a function to retrieve the value of a variable.
<code>setter</code>	Declares a function to set the value of a variable.
<code>add_differential_variable</code>	Registers a differential variable (and its derivative) with the solver instance, specifying getter / setter functions and dimensions.
<code>add_algebraic_variable</code>	Registers an algebraic variable with the solver instance, specifying getter / setter functions and dimensions.
<code>add_equation</code>	Registers an equation with the solver instance.
<code>residual_function</code>	Declares the residual function for an equation, representing the difference between its sides.
<code>pder_function</code>	Declares the partial derivative function for an equation with respect to a variable.
<code>set_residual_function</code>	Associates a residual function with a registered equation.
<code>add_pder_function</code>	Associates a partial derivative function with a registered equation and independent variable pair.

the MARCO runtime libraries – in order to decouple future evolution of the API from the dialect, minimize the amount of compiler-generated code, and introduce additional computational logic besides the one already implemented by IDA. The `ida` dialect is used to interface with such wrapper. An initial implementation of this MLIR dialect was presented by Camillucci [18]. The approach described in this manuscript refines and extends that work, offering a more seamless integration with the overall compilation pipeline. By introducing higher-level abstractions, the proposed dialect eliminates the need to directly manage low-level details of the solver API, such as explicit memory handling, thereby improving both modularity and maintainability.

The most relevant operations of the redesigned `ida` dialect – summarized also in Table 3 – consist in the following. The `instance` operation declares an instance of the solver, enabling multiple instances to coexist (e.g., to support separate configurations for different phases of the simulation). Each instance is initialized through the `init` operation, which allocates the memory for the instance of the solver. The initialization must be performed before any other operation of the dialect is invoked on the instance. Each instance of the IDA solver is then informed about the set of differential and algebraic variables through the `add_differential_variable` and `add_algebraic_variable` operations. Derivative variables are specified implicitly when adding the corresponding differential variable. Both operations are decorated with attributes that specify the dimensions of the variable – possibly empty, in the case of scalars – as well as the `getter` and `setter` functions needed by the runtime environment to access it. Getter and setter functions are declared using the `getter` and `setter` operations.

Listing 6 shows how these operations can be used with respect to the model of Listing 3 (which, if ignoring the materialization of derivatives, is fully reported in Listing 2). 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). Moreover, as anticipated, the information about the `der_T` variable is already provided by `add_differential_variable` with the `T` variable.

Listing 7 shows how equations are represented in the `ida` dialect. Each equation that is handled by the solver – in this example, only one – is described through an instance of the `add_equation` operation, taking the reference to the solver instance and the attribute space describing the iteration space of the equation. The attribute space = `#modelica<indices [2, 999]>` specifies that the induction variable – (`%i`) in Listing 3 – can, and will, assume the values from 2 to 999.

```

module {
  bmodelica.model @Thermal1D attributes {
    der = [#bmodelica.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
    } : !ida.variable

    func.return

  }

  ida.getter @T_getter(%i: index) -> f64 {
    %0 = bmodelica.variable_get @Thermal1D::@T : tensor<1000xf64>
    %1 = tensor.extract %0[%i] : tensor<1000xf64>
    ida.return %1 : f64
  }

  ida.setter @T_setter(%v: f64, %i: index) {
    %0 = bmodelica.variable_get @Thermal1D::@T : tensor<1000xf64>
    %1 = tensor.insert %v into %0[%i] : tensor<1000xf64>
    bmodelica.variable_set @T, %1 : tensor<1000xf64>
    ida.return
  }

  // Getter and setter for 'der_T' are omitted for brevity.
  ...
}

```

Listing 6. Declaration of variables within the `ida` dialect.

```

module {
  bmodelica.model @Thermal1D attributes {
    der = [#bmodelica.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.variable

    // c * der(T[i]) = g * (T[i - 1] - T[i]) + g * (T[i + 1] - T[i])
    %eq = ida.add_equation @ida {
      space = #bmodelica.indices<[2, 999]>
    } : !ida.equation

    ida.set_residual_function @ida, %eq, @eq_res : !ida.equation
    ida.add_pder_function @ida, %eq, %T, @eq_pder_T : !ida.equation
  }

  ida.residual_function @eq_res(%i: index) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }

  ida.pder_function @eq_pder_T(%i: index) -> f64 {
    %0 = ... : f64
    ida.return %0 : f64
  }
}

```

Listing 7. Declaration of equations within the `ida` dialect.

Additional properties however need to be described for the numerical integration method to work. Specifically, for the BDF method, the behavior of each equation is described by providing its residual function – that is, the difference between its equation sides – and its partial derivatives with respect to each of the variables – that is, the Jacobian matrix. These functions appear as occurrences of the operations `residual_function` and `pder_function`, which are associated with the equations using the `set_residual_function` and `add_pder_function` operations. In MARCO, the partial derivatives are computed by applying the **Automatic Differentiation (AD)** technique [29, 48]. More in detail, forward AD has been chosen for its implementation simplicity, but backward AD may also be applied. To implement forward AD, operations among the various dialects are enriched through the use of an interface from the `bmodelica` dialect that abstracts the AD’s requirements.

5.3 The *runtime* Dialect

The simulation driver – that is, the component controlling the runtime macro-actions – is defined in a separate runtime library to ease the development. However, this driver still has to obtain model-specific information that does not influence the numerical integration algorithm, and that is beyond a mathematical representation of the system. Thus, the simulation driver requires some functions and/or global variables to be generated by the compiler. These pieces of information can be expressed with high-level operations, which are then translated to low-level functions through appropriate conversion passes targeting already existing MLIR dialects.

In MARCO, the `runtime` dialect is defined for this purpose. The semantics of the operations are described in Table 4, while a concrete example is provided in Listing 8. The example contains metadata such as the name of the model being simulated (`model_name`), the number and the name of the variables in use, as well as their dimensionality and size (`variable_dimensions`). Similarly to the `ida` dialect, the `variable_getters` operation lists the functions to be used at runtime to retrieve each

```

module {
  runtime.model_name "Thermal1D"

  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 8. `runtime` dialect exposing the common traits of the simulation of the model represented in Listing 3. A summary of the operation semantics is provided in Table 4.

Table 4. Summary of the Main Operations of the Runtime Dialect

Operation	Purpose
<code>model_name</code>	Specifies the name of the model being simulated.
<code>number_of_variables</code>	Indicates the number of variables in the model.
<code>variable_names</code>	Lists the names of all variables.
<code>variable_dimensions</code>	Describes the dimensions of each variable.
<code>variable_getter</code>	Declares a function to retrieve the value of a variable, optionally indexed for arrays.
<code>variable_getters</code>	Lists the getter functions for retrieving variable values at runtime.

variable value. Such functions are declared using the `variable_getter` operation, which receives the indices of the scalar variable of interest and returns its value.

6 Evaluation

The evaluation of the proposed MLIR-based approach to the problem of compiling DAE models – and of the resulting MARCO compiler – 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.

6.1 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 to reconstruct or analyze these concepts from procedural code, a non-trivial and error-prone task. An example is the reordering of **strongly connected components (SCCs)**, which allows the scheduling algorithm to operate on a single MLIR entity rather than refactoring the whole model code.

Additionally, we argue that it is easier to debug individual compiler passes by comparing different MLIR code printouts – which expose more human-readable semantics – with respect to the low-level IR of LLVM. The addition of new features in traditional non-MLIR compilers is often a very hard task if the originally employed data structures are not appropriate for it. This issue is also present in designs of state-of-the-art compilers such as Clang, which presents interesting engineering challenges [33], whereas the abstractions of MLIR also simplify the refactoring required for supporting additional features.

With respect to the DAE-based languages field, it is worth noting that the Modelica community has been aware of bottlenecks in large-scale model compilation with state-of-the-art solutions – including their open-source flagship compiler OMC – for a long time [19]. Although several past and ongoing efforts [6, 52] have been directed toward this issue, such bottlenecks remain in OMC to this day. By applying a multi-layer approach, MARCO allowed us to easily implement more efficient solutions to the critical problem of *matching* [24] that clearly demonstrated ground-breaking advantages. These gains are confirmed by the charts shown in the next section.

6.2 Quantitative Evaluation of Compiler Performance

The set of Modelica compilers available for a sensible comparison is affected by the age and requirements of some of these tools. We must exclude Dymola, as its licensing terms explicitly forbid benchmarking its performance. JModelica, as a discontinued project, does not have significant support; therefore, the best comparison is performed with the OMC.

We compare MARCO and OMC on two benchmarks that exploit a minimal set of language features while implementing common patterns of DAE-based modeling that are parameterized as function of model complexity (i.e., number of equations). Despite the simplicity of the employed syntax and semantics – which makes them compatible with MARCO – the models under test effectively captures the characteristics of industrially relevant systems, whose simulation is typically performed using spatial discretization techniques such as the employed finite volumes method [57, 58].

The first model describes a silicon integrated circuit, in which a constant power source is continuously applied on half of its bottom surface, and the temperature across the volume of the chip is discretized into a parameterized three-dimensional matrix. This model is built using the object-oriented features of Modelica, leveraging the same physics principles of the more straightforward one-dimensional example of Listing 1. The parametrization on the three spatial axes allows us to simulate the same physical model at an arbitrarily fine degree of granularity, which corresponds to an arbitrarily large number of DAEs in the system.

The second model focuses on the thermal behavior of 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. Due to space constraints, we did not analyze the implementation of user-defined functions in this manuscript, but – apart from language-specific features like dynamic array resizing – they can be easily represented through upstream MLIR dialects, thus not requiring particular extensions. Similarly to the previous model, this benchmark is also parametrized to scale the number of equations in the system by varying the spatial discretization of each heat exchanger, the number of heat exchangers in each group, and the number of groups placed in series.

Each configuration of the models under test has been compiled and simulated using the forward Euler method and the IDA solver. In the case of MARCO, the source model has been compiled and simulated using both an array-based Base Modelica representation and the scalar-based DAE counterpart (which are obtained through OMC by respectively enabling or disabling array preservation during the flattening stage), to highlight the benefits of preserving high-level information throughout the compilation pipeline. Finally, the forward Euler simulations have also been compared to a hand-written C++ implementation of the same models, which serves as a reference for the best performance achievable while still leveraging a reasonably high-level language.

In the case of the forward Euler integration method, a fixed number of timesteps – 266667 for the silicon integrated circuit model and 40000 for the heat exchangers model – has been used for all the tested configurations to ensure numerical stability throughout the whole set of tests. In the case of IDA, the number of steps has instead been left free to vary and decided by the solver itself. All tests have been performed on a dedicated CPU server partition from the IDUN HPC cluster [56] featuring 32 cores of an Intel Xeon Gold 6348 processor running at 2.60 GHz and 128 GB of RAM. The software projects have been built from source code using recent rolling releases. More specifically, MARCO has been built on commit *24f25a9*, MARCO runtime libraries on commit *b142d4e*, and OMC on commit *ge499b2*. The C++ reference implementation has been compiled using gcc 12.2.0. The benchmark suite is also distributed together with this

manuscript and freely available online,¹ and reproducible using containerization technologies. The suite is also comprehensive of the printouts showcasing how the IR of the benchmarked models evolve throughout the compilation pipeline. However, it should be noted that only a subset of the employed operations and transformations are described in this manuscript.

The data of the experiments is shown in Figure 4, in a matrix that places the model (silicon chip or heat exchangers) and the numerical integration algorithm (forward Euler or BDF through IDA) in its rows, and the various measurements in its columns. When preserving array semantics, MARCO significantly outperforms OMC in code size, compilation time, and simulation time throughout all the evaluated configurations.

It appears evident that compilation time – that is, the time taken by the benchmarked tool to generate an executable binary starting from Modelica code, shown in the leftmost column of Figure 4 – is a big issue in OMC, and also in MARCO if array preservation is disabled. With the silicon chip model, the smallest configuration of 1040 equations leads to OMC being 6 times slower than MARCO when selecting the forward Euler method, and 3 times slower when selecting IDA. With both numerical integration algorithms there is a degrading trend. With the biggest model – of approximately 185K equations – that could be successfully compiled by both the tools, MARCO kept a compilation time between 1 and 2 seconds, while OMC reached 14 and 10 minutes for Euler and IDA. Bigger models could be compiled with MARCO within the same amount of time as smaller ones. In contrast, the C++ benchmark maintains a constant compilation time regardless of model size, as the model parameters are specified at runtime and do not require changes to the source code. The different trends between MARCO and OMC are the result of the different software architecture employed in the two solutions, with the multi-level approach easily enabling the preservation of array variables and array equations. In the case of forward Euler algorithm, the causalization process can take advantage of such information [13, 24, 40, 41] and avoid the scalarization approach that instead characterizes OMC. Similarly, in the case of IDA, the functions computing residual values and partial derivatives of equations are declared at the array level. The effect of the newly employed IRs is demonstrated by also feeding the scalarized version of the model to MARCO. In this situation, no array semantics are present within the model, the array-aware optimizations can not be applied, and the compilation time therefore exhibits a scaling behavior similar to OMC.

It must be noted that, to enable a fair comparison between the two tools, the compilation time indicated for MARCO is comprehensive of the pre-processing time spent by OMC to instantiate the model and generate its Base Modelica representation, on which MARCO effectively starts working. This code transformation is currently not performed by MARCO for time-constraints reasons, but nonetheless such a feature could be implemented in the future.

The same trends shown in the compilation time are reflected in the code size measurements, shown in the middle column of Figure 4. On the MARCO side, the chosen IRs are represented by the Base Modelica dialect – described in Section 5.1 and obtained right after the conversion of the Base Modelica code into its MLIR counterpart – and the LLVM-IR code obtained as a result of the MLIR translation pipeline implemented in the frontend of MARCO. The sizes for `bmodelica` and LLVM-IR are respectively measured as their on-disk textual and bitcode dumps. On the OMC side, the measurement has been performed on the textual representation of the C files generated by OMC – the only measurable IR emitted by OMC during the compilation. IR and binary sizes in MARCO manifest constant trends while increasing the size of the model, similar to what could be expected by the C++ implementation, while OMC exhibits a linear increase in the produced code. Although subsequent compiler transformations may counterbalance the code size growth, this trend reaches unfeasible levels before the standard gcc or LLVM optimizations can be applied. The

¹<https://github.com/modelica-polimi/marco-dialects-paper-benchmarks/>

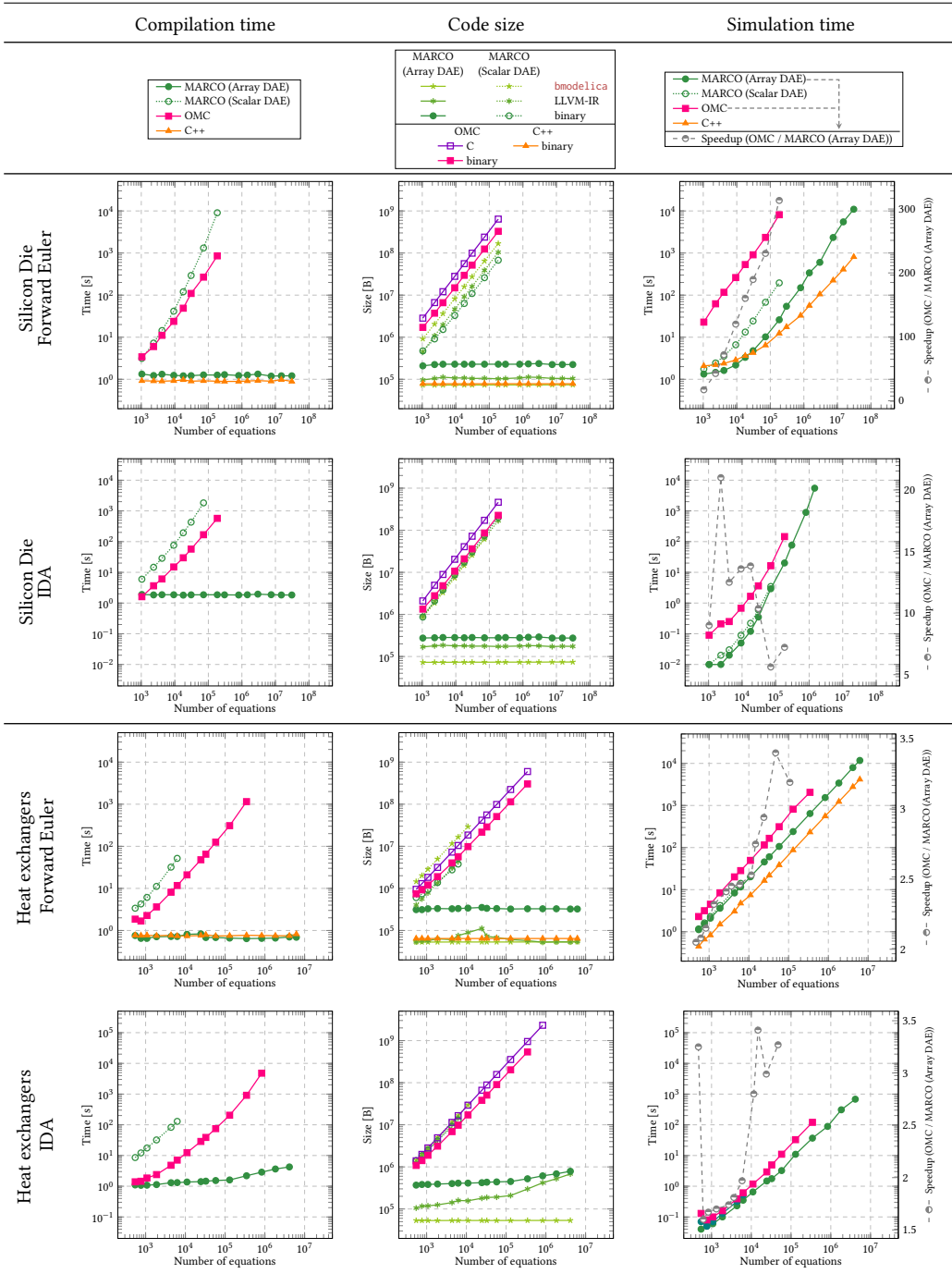


Fig. 4. Performance comparisons of MARCO and OMC, in terms of generated code size, compilation time, and simulation time, with different models and solvers. In all experiments, the compilation and simulation tasks are limited to 5 hours.

ablation study with the scalarized version of the model again confirms that the array-preserving approach is the key to obtain constant code sizes. The array-aware approach of MARCO produces a final binary of approximately of 218KB and 268KB for Euler and IDA, whereas the results of OMC reach 312MB and 215MB at $\sim 185K$ equations.

Finally, let us analyze the time taken by the generated binary files to perform the simulation of the dynamic behavior of the models, shown in the rightmost column of Figure 4. Trends are similar between the binaries generated by MARCO and OMC. This is expected since the computations, the underlying hardware, and the data – both input and output data – are identical. A comparison between the array-preserving and scalarized versions compiled with MARCO also confirms that array-aware optimizations yield equal or better simulation times, thanks to improved memory access patterns and reduced cache misses. Compared to OMC, the array-preserving model compiled with MARCO is also consistently faster – between 5x and 300x depending on the configuration – with respect to the one compiled with OMC. An increasing gap is nonetheless visible when comparing both the implementations against the C++ version, and mandates future investigations regarding runtime aspects (e.g., cache optimizations).

All the trends described are not impacted significantly by the model choice (i.e., they are the same for both the silicon die and the heat exchangers models), with MARCO outperforming OMC in all the considered metrics when array semantics are preserved. A little degradation in the performance of MARCO can be observed when using IDA, due to the sparse accesses performed by the equations. These sparse accesses translate into an increasing amount of callback functions to be given to the runtime environment. These callbacks are needed for the DAE-specific optimizations described in [55], which need to obtain information about the possibly non-zero partial derivatives and access patterns of the involved variables. Nonetheless, the performance of MARCO remains significantly better than OMC in all the considered metrics, and this last inefficiency can be addressed in future works by improving serialization capabilities.

7 Conclusions

We have presented three MLIR dialects that are used within the MARCO compiler for the compilation of a DAE model written in Base Modelica. These dialects represent the backbone of a modular and scalar approach to design a compiler for DAE-based models and thus enable the scale up of digital simulations in multiple engineering domains. We reserved special attention to array variables and array equations, whose preservation throughout the compilation pipeline has been previously identified as fundamental for large-scale modeling [8]. The ablation study in Section 6 demonstrated that the observed performance gains stem from the proposed IR design principles – able to capture and propagate array semantics throughout the pipeline – rather than being an inherent feature of upstream MLIR itself. Nonetheless, the MLIR technology represents the key enabler for their implementation through its modular and extensible architecture, thus effectively making the subsequent optimizations [24, 40] possible.

Future works include the extension to support a broader range of Base Modelica features, such as hybrid systems modeling. Indeed, Modelica allows for the modeling of digital components as well, a trait that enables the construction of digital twins of cyber-physical systems. This feature requires conditional expressions – *time events* and *state events* – to be represented in an efficient way within the compiler.

Additionally, we aim to enhance our benchmark coverage by supporting the ScalableTestSuite project [19], which aligns well with the large-scale capabilities of MARCO. Expanding the set of available numerical integration algorithms is also a priority.

From a performance perspective, the rich MLIR environment opens up the option to use these abstractions to leverage heterogeneous accelerator architectures such as GPUs. In this direction,

two initial objectives are the proper identification of the parallelizable sets of equations, and the design of an intermediate representation capturing such information.

Acknowledgments

The authors also want to thank all the contributors to the project, and professor Francesco Casella for the useful insights on the modeling domain.

References

- [1] MLIR: Documentation. Retrieved July 19, 2026 from <https://mlir.llvm.org/docs>
- [2] 2019. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law. Retrieved July 19, 2026 from <https://research.google/pubs/mlir-primer-a-compiler-infrastructure-for-the-end-of-moores-law>
- [3] Modelica: Language Reference. Retrieved July 19, 2026 from <https://specification.modelica.org/>
- [4] Torch-MLIR. Retrieved July 19, 2026 from <https://github.com/llvm/torch-mlir>
- [5] IREE. Retrieved July 19, 2026 from <https://iree.dev>
- [6] Karim Abdelhak, Francesco Casella, and Bernhard Bachmann. 2023. Pseudo array causalization. In *Proceedings of the 15th International Modelica Conference*. 177–188. DOI : <https://doi.org/10.3384/ecp204177>
- [7] Shirang Abhyankar, Jed Brown, Emil M. Constantinescu, Debojyoti Ghosh, Barry F. Smith, and Hong Zhang. 2018. PETSc/TS: A modern scalable ODE/DAE solver library. arXiv:1806.01437 [math.NA]. Retrieved from <https://arxiv.org/abs/1806.01437>
- [8] Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. 2019. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference* (03 2019), 313–320. DOI : <https://doi.org/10.3384/ecp19157313>
- [9] Giovanni Agosta, Francesco Casella, Daniele Cattaneo, Stefano Cherubin, Alberto Leva, Michele Scuttari, and Federico Terraneo. 2023. MARCO: An experimental high-performance compiler for large-scale modelica models. In *Proceedings of the 15th International Modelica Conference*. Linköping Electronic Conference Proceedings. DOI : <https://doi.org/10.3384/ecp20413>
- [10] Nicolas Bohm Agostini, Serena Curzel, David Kaeli, and Antonino Tumeo. 2022. SODA-OPT an MLIR based flow for co-design and high-level synthesis. In *Proceedings of the 19th ACM International Conference on Computing Frontiers (CF'22)*. ACM, New York, NY, USA, 201–202. DOI : <https://doi.org/10.1145/3528416.3530866>
- [11] Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. 2009. JModelica — an open source platform for optimization of modelica models. In *Proceedings of the 6th Vienna International Conference on Mathematical Modelling*; Conference date: 11-02-2009.
- [12] Mats Andersson. 1989. An object-oriented language for model representation. In *Proceedings of the 2nd IEEE Control Systems Society Workshop on Computer-Aided Control System Design*. 8–15.
- [13] Matthias Arzt, Volker Waurich, and Jörg Wensch. 2014. Towards utilizing repeating structures for constant time compilation of large Modelica models. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'14)*. ACM, New York, NY, USA, 35–38. DOI : <https://doi.org/10.1145/2666202.2666207>
- [14] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A survey on reactive programming. *ACM Computing Surveys (CSUR)* 45, 4 (2013), 1–34.
- [15] Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. 2019. A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access* 7 (2019), 167653–167671. DOI : <https://doi.org/10.1109/ACCESS.2019.2953499>
- [16] Paul I. Barton and Constantinos C. Pantelides. 1994. Modeling of combined discrete/continuous processes. *AIChE Journal* 40, 6 (1994), 966–979. DOI : <https://doi.org/10.1002/aic.690400608>
- [17] Nick Brown, Maurice Jamieson, Anton Lydike, Emilien Bauer, and Tobias Grosser. 2023. Fortran performance optimisation and auto-parallelisation by leveraging MLIR-based domain specific abstractions in Flang. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 904–913.
- [18] Nicola Camillucci. 2022. Compilation and Optimization of Large Scale Modelica DAE Models. Retrieved July 19, 2026 from <https://www.politesi.polimi.it/handle/10589/186108>
- [19] Francesco Casella. 2015. Simulation of large-scale models in Modelica: State of the art and future perspectives. In *Linköping Electronic Conference Proceedings*. 459–468.
- [20] François E. Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer Science & Business Media.
- [21] François E. Cellier, Matthias Krebs, and AG DaimlerChrysler. 2007. Analysis and simulation of variable structure systems using bond graphs and inline integration. *Simulation Series* 39, 1 (2007), 29.

- [22] Mike Dempsey. 2006. Dymola for multi-engineering modelling and simulation. In *Proceedings of the 2006 IEEE Vehicle Power and Propulsion Conference*. 1–6. DOI : <https://doi.org/10.1109/VPPC.2006.364294>
- [23] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as hardware compiler infrastructure. In *Proceedings of the Workshop on Open-Source EDA Technology (WOSET)*.
- [24] Massimo Fioravanti, Daniele Cattaneo, Federico Terraneo, Silvano Seva, Stefano Cherubin, Giovanni Agosta, Francesco Casella, and Alberto Leva. 2023. Array-aware matching: Taming the complexity of large-scale simulation models. *ACM Transactions on Mathematical Software* 49, 3, Article 27 (Sep 2023), 25 pages. DOI : <https://doi.org/10.1145/3611661>
- [25] Jens Frenkel, Günter Kunze, Peter Fritzon, Martin Sjölund, Adrian Pop, and Willi Braun. 2011. Towards a modular and accessible Modelica compiler backend. In *Proceedings of the 8th International Modelica Conference (Modelica'2011)*. Linköping University Electronic Press, 232–238. Retrieved from https://2011.international.conference.modelica.org/proceedings/pages/papers/10_3_ID_110_a_fv.pdf
- [26] Peter Fritzon, Adrian Pop, Karim Abdelhak, Adeel Ashgar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, et al. 2020. The OpenModelica integrated environment for modeling, simulation, and model-based development. *Modelling, Identification, and Control* 4, 4 (2020), 241–285. DOI : <https://doi.org/10.4173/mic.2020.4.1>
- [27] Peter Fritzon, Adrian Pop, and Peter Aronsson. 2005. Towards comprehensive meta-modeling and meta-programming capabilities in modelica. In *Proceedings of the 4th International Modelica Conference*. Citeseer.
- [28] Peter Fritzon, Adrian Pop, Martin Sjölund, and Adeel Asghar. 2019. MetaModelica—a symbolic-numeric Modelica language and comparison to Julia. In *Proceedings of the 13th International Modelica Conference*.
- [29] Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM.
- [30] Adrien Guironnet, Marianne Saugier, Sebastien Petitrenaud, Florent Xavier, and Patrick Panciatici. 2018. Towards an open-source solution using modelica for time-domain simulation of power systems. In *Proceedings of the 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*. DOI : <https://doi.org/10.1109/isgteurope.2018.8571872>
- [31] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. DOI : [https://doi.org/10.1016/S0167-6423\(02\)00109-0](https://doi.org/10.1016/S0167-6423(02)00109-0). Special Issue on Language Descriptions, Tools and Applications (L DTA'01).
- [32] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software* 31, 3 (Sep 2005), 363–396. DOI : <https://doi.org/10.1145/1089014.1089020>
- [33] Gábor Horváth, Péter Szécsi, Zoltán Gera, Dániel Krupp, and Norbert Pataki. 2018. [Engineering Paper] Challenges of implementing cross translation unit analysis in clang static analyzer. In *Proceedings of the 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 171–176. DOI : <https://doi.org/10.1109/SCAM.2018.00027>
- [34] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, et al. 2020. Compiling ONNX neural network models using MLIR. arXiv:2008.08272 [cs.PL]. Retrieved from <https://arxiv.org/abs/2008.08272>
- [35] Gerd Kurzbach, Oliver Lenord, Hans Olsson, Martin Sjölund, and Henrik Tidefelt. 2023. Design proposal of a standardized Base Modelica language. In *Proceedings of the 15th International Modelica Conference*. DOI : <https://doi.org/10.3384/ecp204469>
- [36] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. DOI : <https://doi.org/10.1109/CGO.2004.1281665>
- [37] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. DOI : <https://doi.org/10.1109/CGO51591.2021.9370308>
- [38] Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. 2020. A state-of-the-art survey of Digital Twin: techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing* 31, 6 (2020), 1313–1337.
- [39] Mengnan Liu, Shuiliang Fang, Huiyue Dong, and Cunzhi Xu. 2021. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems* 58 (2021), 346–361. DOI : <https://doi.org/10.1016/j.jmsy.2020.06.017>
- [40] Denise Marzorati, Joaquín Fernández, and Ernesto Kofman. 2024. Efficient matching in large DAE models. *ACM Transactions on Mathematical Software* 50, 3 (Jun 2024) 1–25. DOI : <https://doi.org/10.1145/3674831>

- [41] Denise Marzorati, Joaquin Fernández, and Ernesto Kofman. 2022. Efficient connection processing in equation-based object-oriented models. *Applied Mathematics and Computation* 418 (2022), 126842. DOI : <https://doi.org/10.1016/j.amc.2021.126842>
- [42] Marek Mateják and Jiří Kofránek. 2015. Physiomodel - an integrative physiology in Modelica. In *Proceedings of the 2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 1464–1467. DOI : <https://doi.org/10.1109/EMBC.2015.7318646>
- [43] MathWorks. [n. d.]. *Simscape, Model and Simulate Multidomain Physical Systems*. Online, accessed 2026-07-19.
- [44] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. 1998. Physical system modeling with Modelica. *Control Engineering Practice* 6, 4 (1998), 501–510.
- [45] S. E. Mattsson and G. Söderlind. 1993. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 14, 3 (1993), 677–692.
- [46] Thierry S. Noudui Michael Wetter, Wangda Zuo and Xiufeng Pang. 2014. Modelica buildings library. *Journal of Building Performance Simulation* 7, 4 (2014), 253–270. DOI : <https://doi.org/10.1080/19401493.2013.765506>
- [47] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *Proceedings of the 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. DOI : <https://doi.org/10.1109/PACT52795.2021.00011>
- [48] Richard D. Neidinger. 2010. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review* 52, 3 (2010), 545–563.
- [49] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell'02)*. ACM, New York, NY, USA, 51–64. DOI : <https://doi.org/10.1145/581690.581695>
- [50] Hugo Pompougnac, Ulysse Beaunon, Albert Cohen, and Dumitru Potop Butucaru. 2022. Weaving synchronous reactions into the fabric of SSA-form compilers. *ACM Transactions on Architecture and Code Optimization* 19, 2, Article 22 (Mar 2022), 25 pages. DOI : <https://doi.org/10.1145/3506706>
- [51] Adrian Pop and Peter Fritzson. 2006. MetaModelica: A unified equation-based semantical and mathematical modeling language. In *Modular Programming Languages*. David E. Lightfoot and Clemens Szyperski (Eds.), Springer Berlin Heidelberg, Berlin, 211–229.
- [52] Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, Rüdiger Franke. 2019. A new openmodelica compiler high performance frontend. In *Proceedings of the 13th International Modelica Conference*, Vol. 157. 689–698. DOI : <https://doi.org/10.3384/ecp19157689>
- [53] Sylvain Quoilin, Adriano Desideri, Jorrit Wronski, Ian Bell, and Vincent Lemort. 2014. ThermoCycle: A modelica library for the simulation of thermodynamic systems. In *Proceedings of the 10th International Modelica Conference (Linköping Electronic Conference Proceedings, 96)*. Linköping University Electronic Press, 683–692. DOI : <https://doi.org/10.3384/ECP14096683>
- [54] Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E. Nagel, and Ines Gubsch. 2015. Exploiting repeated structures and vectorization in Modelica. In *Proceedings of the 11th International Modelica Conference*. Linköping Electronic Conference Proceedings. DOI : <https://doi.org/10.3384/ecp15118265>
- [55] Michele Scuttari, Nicola Camillucci, Daniele Cattaneo, Giovanni Agosta, Francesco Casella, Stefano Cherubin, and Federico Terraneo. 2023. Clever DAE: Compiler optimizations for digital twins at scale. In *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF'23)*. ACM, 344–350. DOI : <https://doi.org/10.1145/3587135.3589945>
- [56] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. 2022. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure. arXiv:[1912.05848](https://arxiv.org/abs/1912.05848) [cs.DC]. Retrieved July 19, 2026 from <https://arxiv.org/abs/1912.05848>
- [57] Federico Terraneo, Alberto Leva, William Fornaciari, and David Atienza. 2023. Modeling and simulation challenges and solutions in cooling systems for nanoscale integrated circuits [Feature]. *IEEE Circuits and Systems Magazine* 23, 1 (2023), 36–56. DOI : <https://doi.org/10.1109/MCAS.2023.3234727>
- [58] Federico Terraneo, Alberto Leva, William Fornaciari, Marina Zapater, and David Atienza. 2022. 3D-ICE 3.0: Efficient nonlinear MPSoC thermal simulation with pluggable heat sink models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 4 (2022), 1062–1075. DOI : <https://doi.org/10.1109/TCAD.2021.3074613>
- [59] L. Vanfretti, T. Rabuzin, M. Baudette, and M. Murad. 2016. iTesla power systems library (iPSL): A modelica library for phasor time-domain simulations. *SoftwareX* 5 (2016), 84–88. DOI : <https://doi.org/10.1016/j.softx.2016.05.001>
- [60] Hanchen Ye, HyeGang Jun, Hyunmin Jeong, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A scalable high-level synthesis framework with multi-level transformations and optimizations. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. ACM, New York, NY, USA, 1355–1358. DOI : <https://doi.org/10.1145/3489517.3530631>

Received 18 January 2025; revised 30 April 2026; accepted 15 July 2026