POLITECNICO DI MILANO
Master's degree in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

M.A.R.C.O.: an experimental high-performance Modelica compiler for large scale systems

Relatore:
Giovanni Agosta

Correlatori:
Francesco Casella
Daniele Cattaneo
Stefano Cherubin
Alberto Leva
Federico Terraneo
Silvano Seva

Tesi di Laurea di:
Massimo Fioravanti, 921149

Academic year 2019-2020

Contents

List of Figures

List of Algorithms

Abstract

Large-scale physical simulations are often composed of multitudinous equations that, in chunks, share the same structure. Furthermore, if the model is provided as a system of equations written in a domain specific language, it might be possible to extract a high-level structure and use it to reduce the complexity of translating the model into a computable program.

Still, generic solvers are usually unaware of such high-level features and cannot make proper use of them. All equations are handled independently from each other. Thus the time required to produce a computable simulation starting from a model is tied to the number of equations rather than the number of equations with different structures. When the size of the simulation increases, the amount of time and memory required quickly becomes unreasonable. Furthermore, if a simulation is finally obtained, the code that describes it is often low-performing, due to the disregard of data locality and cache friendliness.

In this document, we inspect Modelica, a domain specific language that offers language features which can be used to vastly improve the compilation performance. We provide formal proof of the complexity of each step of the compilation pipeline with respect to the number of equations of the input model. We describe a subset of the language and algorithms that, given a particular high-level structure, are able to produce a simulation in constant time. Finally, we introduce an implementation for such language subset that, given a fixed high-level structure, compiles in constant time regardless of the size of the simulation and yields much smaller binaries and performs faster than the current state-of-the-art tools.

Sommario

Simulazioni di sistemi fisici su larga scala sono solitamente composti da una multitudine di equazioni le quali, in blocchi, condividono la stessa struttura. Inoltre, se il modello di tale simulazione è fornito come un sistema di equazioni scritte in un qualche linguaggio specifico al problema, potrebbe essere possibile estrarre una struttura di alto livello e usarla per ridurre la complessità di tradurre il modello in una programma computabile.

Solver generici sono solitamente inconsapevoli dell'esistenza di questa struttura ad alto livello e non possono farne un buon uso. Tutte le equazioni sono trattate indipendentemente. Ne consegue che il tempo richiesto per produrre la simulazione di un certo modello è legato al numero di equazioni, anzichè al numero di equazioni con struttura diversa. Quando la dimensione della simulazione aumenta, il tempo e la memoria diventano rapidamente eccessive. Inoltre, se una simulazione è finalmente ottenuta, il codice che la descrive è spesso poco performante, per via della bassa importanza che è stata data alla località dei dati.

In questo documento ispezioniamo Modelica, un linguaggio di modellazione di sistemi fisici che offre caratteristiche adatte ad essere sfruttate per migliorarne la compilazione. Offriamo prova formale della complessità di ogni step della pipeline di compilazione rispetto al numero di equazioni del modello in input. Descriviamo un subset del linguaggio e algoritmi che, per una particolare struttura ad alto livello del modello in input, produce simulazioni in tempo costante. Infine, introduciamo una implementazione per tale subset del linguaggio che, data una particolare struttura ad alto livello, compila simulazioni in tempo costante e produce eseguibili molto più piccoli e più veloci dello stato dell'arte corrente.

Acknowledgments

I would like to extend my thanks and apologies to those that have somehow improved this document.

A special thanks goes to the many names that appear on the cover of this thesis, since without them this document would have been even more it convoluted than currently is. A special apology goes to Federica Gianotti and Pietro Ghiglio that after having read this document are no longer able to solve systems of equations without applying graph homomorphisms to them. A special thanks goes to Alì El Wahsh and Claudia Conchetto to have listened to me while I was complaining about writing this document. A special apology goes to Marina Nikolic and Michele Scuttari that will have the pleasure of maintaining the code base after my departure. And finally a special thanks goes to my family, which incubated a human being for 24 years before he was able to produce something of value.

Chapter 1

Introduction

When the many are reduced to one,
to what is the one reduced?

Koan/Haskell Programmer

1.1 Models and simulations

A world of equations It has been suggested that one of the greatest intuitions of mankind was to notice that there exists a language able to describe nature: we call it math, and as it is indeed astonishing that similar mathematics applies so well to planets and clocks.

The mathematical description of how stars and cogs move is said to be a model. Once a model of a physical phenomenon has been found, then it becomes possible to deduce both the past and the future of such a physical system. Given the power provided by producing such model, it is not surprising that many scientists and engineers are tasked with their creation and analysis. They are called modelers, and the mathematical tool mostly used comes in the form of a system of equations.

Unfortunately, equations-based models are not simulations of the phenomena, but they hold the necessary information to produce such simulations. The task of obtaining the description of a simulation that a computer can execute is a much different kind of task than the one of obtaining a model. Automatically obtaining a simulation from a mathematical model is the concern of this document.

The twin tongues of science and machine The complexity of translating a model into a simulation lies in the different nature between the language of science, that is math, and the language of machines. The first is intended to be of purely declarative - the order of equations is irrelevant. Similarly, it is irrelevant the order in which the terms appear within the equation. A model is meant to be interpreted by the reader, and it is up to the reader to be able to manipulate the equations to produce a meaningful result.

Algorithms, the language one must use to provide a simulation to a computer, are something completely different. A machine expects a series of instructions that allows it to understand how its internal state must be updated and, if the instructions are correct, then the machine will yield the final state of the simulation we intended to run. No equation exists, only a huge number of sequential atomic operations.

Thus, if we wish to obtain a simulation from a model, a translation from pure math to an algorithm is required. Programs that perform such kind of translations are called compilers.

Compilers are tools born from a long tradition, many of the steps they perform are well known. We know how to parse a program from text and represent it in memory, we know how to describe the optimizations that may be applied to them, and, given a high-level algorithm, we know how to yield machine code that is better than the one a human would produce by hand.

Yet, open questions still exist: which is the most performing machine code we can yield when compiling a program that is written in a completely abstract language such as math? How long does it take to perform such a translation? Is it possible to perform such translation in sub-linear time? Can we ensure that the code produced is at least as good as the one a human would yield?

A need for speed The quest for performances brings us to consider questions that may be unexpected: suppose we were able to formally prove that a particular language cannot be compiled in a reasonable amount of time, or that the code generated is, by some metric, worst than the optimal one. What can we do? Should we optimize the average case hoping that it is enough for the average user? Should we abandon the language in favor of another one we can prove to be performant? Should we restrict the language, dropping those features that violate the requirements?

These are the questions we needed to answer in our research, so that it may have been possible to provide the implementation of a compiler able execute its own job efficiently.

1.2 Modelica

Our attempt of creating such a compiler is not the first in the domain of modeling. There already exists other languages that have been designed for such tasks. We focus on one of these languages, named the Modelica language, since it is among the most used language in its own domain.

The purpose of Modelica is exactly the one described earlier: it allows to provide a set of equations and tries to translate it into a simulation.

Unfortunately, Modelica suffers from performance issues, the time needed to produce a simulation is a function of the number of equations involved into the system, even if such equations respect some strict pattern that a human may exploit to produce the simulation in a shorter time. Such limitations prevent any meaningful usage of the language in the domain of large scale systems.

Re-purposing a language Our interest is, therefore, creating a compiler for the Modelica language that is able to produce simulations as quickly as possible, while providing a language as simple as possible, while still usable. We wish to find and retain all language features that do not affect the compilation time, and to offer an alternative for those that do.

Furthermore, we wish to formally prove that some language features are not compatible with fast compilation, and to provide a framework in which to analyze and describe the issues tied to the compilation speed.

Finally, we wish to provide a new Modelica compiler, built on the latest compiler technology, to become a useful tool and library that can be used to produce simulations.

1.3 Results

In this document we provide:

Proof of complexity for the pipeline stages needed to produce a simulation based on the Euler method. We prove that the unbound language cannot be compiled in constant time, and thus it must be restricted.

A subset of Modelica able to preserve arrays across the compilation pipeline we discuss a subset of the language that is powerful enough to be useful and at the same time retains the ability of allow constant time compilation for a given high-level structure. It involves novel algorithms and a novel representation of the in-memory data structures.

A novel implementation We provide an implementation of these algorithms, as well as a lowerer toward LLVM-IR, so that our experimental compiler can be compared with the current state-of-the-art tools.

Chapter 2

State Of The Art

Whatever it is, it can't be that bad.

Londo Molari - Babylon 5

This document focuses on the domain specific languages used for physical model design, and, among those, the Modelica language in particular. It is likely that readers of this documents will either have a background in computer science or physical systems modeling, not both, and thus the challenges that each field has to handle may be unknown to the other.

For this reason, this chapter will provide the state of the art of Modelica and its main open source implementation. It includes information regarding compilers, most common compiler frameworks, and some compilation techniques that are usually invisible to the user. Furthermore, it presents the differences between domain specific languages for physical modeling and regular general-purpose programming languages.

Finally, it must be noticed that both *Modelica* and *LLVM*, the language we implement and the compiler framework we use as a back end, are not core to the results of this document. We will provide proofs of complexity and algorithms that can be used for any domain specific language similar to *Modelica* and we may have targeted any compiler framework beside *LLVM*.

2.1 Modeling

The purpose of a model is to describe the representation of a physical phenomena as a mathematical object. Such object is often a system of equations. If the phenomena is static then there exists a finite amount of solutions and there is no need for time to be a variable in the system of equations. Otherwise some or all variables may be time-varying, that is, they are function of time. If such is the case then in the system of equation even derivative terms may be present.

Furthermore, an equation of the system may involve variables at different time instant. This is the case of equations describing the evolution of the system over time, an example is $a(t) = b(t-1)$. When solving a time-varying systems at some instant of time t we may need to have already calculated some variable at previous time instants.

2.2 System of equations

DAE and ODE A system of equations is a set of equations and we denote it as E. These equations involve variables $x(t) = (x_1(t), \ldots x_n(t))$, where each $x_k(t)$ is the value of the variable x_k at time t.

Equations may involve derivatives of variable as well. We denote the derivative respect to time of variable x as $\dot{x}$.

Therefore, each equation f may involve a variable x, its derivative, or time t. Thus, a system of equations can be formulated as

$$F(x, \dot{x}, t) = 0$$

Equations in this form are called differential-algebraic system of equations *(DAEs)*. It may be possible to write a DAE system as:

$$\dot{x}(t) = F(x(t), t)$$

That is, it is possible to express analytically the $\dot{x}$ term. If it is then it is called a system of ordinary differential equations (ODE).

ODE systems are, in general, easier to solve than DAE and since the derivatives are explicit, it is possible to make $\dot{x}$ explicit.

Explicitating $x'(t)$ and Euler Method When it is not possible to obtain ODE system from a DAE system from exact methods, it may be possible, to obtain it from approximation. Many such approximations exists, and since they are not the focus of this work we only present the Euler method. Such method states that a equation written as

$$y'(t) = f(t, y(t)) \quad y(t_0) = y_0$$

can be rewritten as:

$$y_{n+1} = y_n + h * f(t, y)$$

where h is the time delta. This arises trivially from the definition of derivative.

Domain specific modeling languages There exists different programming languages that allow to provide a declarative descriptions of equations and yield a simulation of such system. Modelica [8], is one this. Modelica is not a implementation but rather it is a standard. We will describe more in depth Modelica features in the next section.

There exists some close-source implementations of *Modelica*, such as *Dymola. Modia* [9] is a open-source alternative to Modelica. While it shares many different features with Modelica it is not a implementation of the standard and it is tied to the *Julia* language.

There exists other more specific languages used to describe system of a particular engineering field. While they may be more performant, such languages are not have a purpose general enough to be of hour interest. Since *Modelica* is the most used among the described languages, we will focus on it.

2.3 Modelica Language

Modelica is an object-oriented, declarative, multi-domain modeling language for component-oriented modeling of complex systems [8]. At an extremely high-level, components are essentially sets of equations, which can be connected to other components by asserting an equality between input and output variables. The task of a Modelica compiler is to produce an executable that performs a correct simulation of the system.

Modelica has many language features that operate at different stages of the compilation pipeline. Many of those features consist of additional syntax that simplifies the process of writing equations, allow to divide components in sub-components or introduces simulation time semantic orthogonal to the topics of this paper. Either because they are high-level features that have been already simplified in earlier stages of the compilation pipeline or because they are used in later stages and are ignored in the stage considered. Therefore, we first introduce only the subset that contains variable definitions, the already mentioned equations and the for-loop syntax.

Consider the following snippet of Modelica code, implementing a component with eight inputs and one output, computed as the average of the sums of each pair of inputs:

Algorithm 1: Example Modelica component

```
model AverageOfSum
  Real inputs[8];
  Real output;
  Real state[4];
 equation
  output * 4 = state[1] + state[2] + state[3] + state[4]; //eq1
  for i in 1:4 loop
   input[2*i] + input[(2*i) + 1] = state[i]; //eq2
  end for;
end AverageOfSum;
```

The snippet key language features are equivalent to those described in the earlier subsections.

- Equations do not describe assignments. Thus, when compiling, simulation equations must be transformed in accordance with their semantic.
- For loops do not describe control flow. Instead, they are resolved at compile-time, and represent a form of meta programming in the Modelica language.
- The top-level model can be composed of fundamental types or of other components, and the top-level model must be simulatable. That is, it must be possible to determine the value of each variable by manipulating the equations.

In the snippet, *AverageOfSum* only provides 5 equations and declares 13 variables, therefore the hierarchy that contains *AverageOfSum* must provide at least 8 more equations. An example of this hierarchy is:

This two pieces of code will be our running example in the introduction.

Algorithm 2: Example of a Modelica top-level component

```
model OuterComponent
  AverageOfSum inner;
 equation
  for i in 1:8 loop
  inner.input[i] = i * i;

end OuterComponent;
```

2.3.1 Modelica Compilation

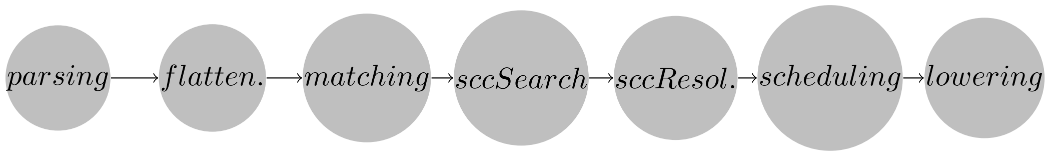


Figure 2.1: Pipeline structure of a classical Modelica compiler, comprising of several different successive stages.

A Modelica compiler is typically structured into a series of stages, as it is shown in figure 2.1 After the source file has been parsed, all the object-oriented feature are simplified. After this step no object oriented language feature is preserved and we only retain a list of scalar variables and a list of scalar equations. We call this step *flattening.*

Afterwards, each equation is matched with a single scalar variable. This step determines which equation will be used to compute the value of which variable. This analysis is known as *matching.* Notice that during the matching step all equations from the whole system contribute to the solution, not only those in a component.

Consider again the running example. As we said *AverageOfSum* cannot be used as a stand-alone component, because there would not be enough equations to calculate all the variables. AverageOfSum must be used by another component that contributes with the required equations. Therefore, our top-level component is *OuterComponent.* One possible matching for the previous example would be the one that associates *output* with *eq1*, and each *state* with the member of *eq2* that uses it. Therefore the equations might be rewritten as:

Algorithm 3: Unrolled Outer Components

```
  // Matched with output
  output/4 = state[1] + state[2] + state[3] + state[4];
  state[1] = input[1] + input[2];  // Matched with state[1]
  state[2] = input[3] + input[4];  // Matched with state[2]
  state[3] = input[5] + input[6];  // Matched with state[3]
  state[4] = input[7] + input[8];  // Matched with state[4]
```

After the matching step, the compiler finds the dependencies among variables. In the running example, output cannot be computed until all elements of *state* have been computed. If *eq1* depended on output as well, then there would have been a cyclic dependency. Cyclic dependencies prevent the compiler from producing a simulation.

Therefore, each equation e will be scheduled after all equations matched to the variables in the right hand of e. This dependencies can be described as a directed graph. When operating on this graph, the problem of finding cyclic dependencies is equivalent to the one of searching strongly connected components (SCC), thus we call this step *SCC search.* If a cyclic dependency was found then we need to perform an *SCC resolution* step. Such step may depend on the resolution technique chosen and it is common to all executions of the compiler.

The last step performed by the compiler is the scheduling. Once we know that there are no cyclic dependencies, we can find an execution order such that all the dependencies are respected. One possible scheduling for our example is:

Algorithm 4: Scheduled Outer Component

```
state[1] = input[1] + input[2];
state[2] = input[3] + input[4];
state[3] = input[5] + input[6];
state[4] = input[7] + input[8];

output / 4 = (state[1] + state[2] + state[3] + state[4]);
```

Now the equations set can be lowered to assignments and finally compiled. In figure 2.1 we show the pipeline of the compiler we described.

2.3.2 Object Oriented Features

The Modelica specifications [3] define the semantics of class extensions, which is a core feature of programming languages. Notice that classes are very different from common object oriented class extensions. Classes do not survive the flattening stage, and are not needed to produce a correct simulation, and do not entail a locality of the fields in memory. Thus, the semantic of class extension is in - its simplest form - just code reuse. Furthermore, Modelica class extensions allow for base class modifications, as an example, consider the snippet of code taken from the Modelica documentation [5] shown in algorithm 5

Algorithm 5: Object Oriented Example

```
 partial model BaseCorrelation
   input Real x;
   Real y;
end BaseCorrelation;

model SpecialCorrelation
    extends BaseCorrelation(x=2);
equation
     y=2/x;
end SpecialCorrelation;
```

The specialized class is allowed to specify the value of base class members, as well as to add new equations to the equation set. Furthermore, the class of any member object of the base class can be replaced as well. x could have been replaced with a custom float type. This behavior is clear in the case of extension, since the extended class will contain the replaced type. Beside violating best practices as Liskov substitution principle [16], it prevents the compiler to detect type correctness by just comparing types, since they may have been modified. Furthermore, with the use of the keyword *replaceable*, this substitutions can be performed on classes that are being used as members rather than being extended. An example is shown in algorithm 6:

Algorithm 6: Repleacable Example

```
 partial model Base
   redeclarable input Real x;
   Real y;
end BaseCorrelation;

model Contained
    BaseCorrelation o(redeclare x=2);
equation
    o.y=2/o.x;
end SpecialCorrelation;
```

This implies that Modelica classes layout can be calculated only at each object declaration site, and that 2 object with the same type may have very different layouts. Thus, Modelica classes are not the definitions of a datatype, rather they are the template used to compute the data layout at each instantiation point.

2.3.3 Flattening and Flat Modelica

As we showed, Modelica object oriented language features must be simplified up to a point that no high-level features are preserved. The Modelica specifications [3] describe the exact rules this flattening process must follow. In particular they specify that each variable and equation of the system must be scalarized and unrolled, until all that remains is a list of scalar variables and single equations. Therefore, there is a clear distinction between the front-end and the back-end of a Modelica compiler. The first handles the syntactic-sugar, code expansion and the production of the flattened list of equations. The second one is tasked to bring that list in a executable form.

Flat Modelica Due to this strong distinction between front-end and back-end a specification for a flat Modelica language is being developed. Such language is intended to establish a layer of compatibility across tools of the Modelica ecosystem.

2.3.4 Limitations of the language

Consider the equations described by the following for-equation:

```
for i in 1:k loop
  input[2*i] + input[(2*i) + 1] = state[i];
```

As we said such loop must be unrolled in the flattening stage. If such is the case then the compile-time will depend on the k compile time constant. Thus, the compilation time will depend not on the length of source file but on the scalar-equations count. Therefore, if a compiler is compliant with the specifications and it unrolls every for-equation, then large simulations cannot be elaborated in reasonable time.

For-loop preservation Due to the reasons shown we would wish to be able to preserve some kind of high level information after the flattening stage. The simplest choice is to preserve the for-equations syntax across front-end and back-end. Many high level features can be transformed into for-loops in the front-end and for-loops are implementable efficiently into machine code, if they can be preserved up to that point.

The flat-Modelica language will include notations for the for-equations as well.

2.3.5 Modelica compilers implementations

Modelica specifications only require that the output of a compiler pass is a simulation. How such simulations is not defined. The main open-source implementation of Modelica is *open Modelica compiler, OMC* [6]. *open Modelica compiler* translates a Modelica simulation into *C* code. The output of OMC must be then compiled by a third-party *C* compiler, such as *GCC*.

OMC is written in Metamodelica [17]. Metamodelica is a Modelica super-set able to express more general programming language features, such as exceptions. *OMC* is therefore a bootstrapping compiler. *OMC* is the only implementation of a Metamodelica compiler.

2.4 Programming VS modeling languages

We provide in table 2.2 the characteristics of the *C* language, as a representative of imperative languages, compared to the *Modelica* language, as a representative of modeling languages.

	C	**Modelica**
source code size tied to compilation time	yes	yes
for-cycles indicies tied execution time	yes	yes
for-cycles indicies tied compilation time	no	yes
array sizes tied to execution time	yes	yes
array sizes tied to compilation time	no	yes
compile time evaluated code	preprocessor only	yes

Figure 2.2: Comparison between C, an imperative programming language, and Modelica, a modeling language. For simplicity, the imperative programming capabilities of Modelica are ignored.

Here we can notice the issue with modeling domain specific languages. By increasing the size of vector or of induction variables the compilation time increases. Beside being unexpected for those who come from the field of computer science, this is a critical issue. If we wish to enable the usage of these domain specific languages for large scale systems this is unacceptable. Input programs of fixed file length quickly becomes too large to be compiled.

We will later prove that one obstacle to fast compilation is the requirement of *Modelica* to allow arbitrary expressions evaluation at compile time.

2.4.1 In-memory representation

We provide now a insight in how compilers handle the compilation of programming languages, so that it may be later be easier to state explicitly what is the issue with the state of the art in the event that a reader may have no deep knowledge in the field.

Formal languages are usually defined by two components, their semantics and their syntax. Semantics is used by the compiler to produce the executable described by that program, while syntax is used to parse the source file and to build a data structure that will be manipulated across the stages of the compilation pipeline. The concrete syntax of a language is the syntax actually used by the user to write a source file and is usually defined by some meta-syntax, such as *EBNF* [14].

Once the source file has been parsed, the textual nature of the concrete syntax is usually lost. A representation equivalent to the source code is held in memory as an abstract syntax tree, where each node of the tree has been generated by a rule of the grammar. As an example, consider a simple language composed only by additions and multiplications that follows the normal precedence rules:

$$ADD := MULT|(MULT + ADD)$$

$$MULT := "term"|("term" + MULT)$$

And the input string $a + b * c + d$.

The abstract syntax tree obtained is shown in Figure 2.3.

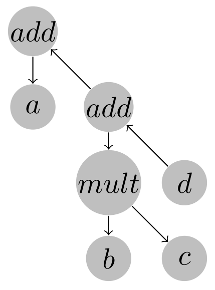


Figure 2.3: Example of Abstract Syntax Tree obtained from parsing a simple mathematical expression

The existence of abstract syntax tree allows us to express the recursion present in the grammar of all programming languages.

If there is no loss of data in the parsing process, then it is possible to dump the abstract syntax tree in textual format. If the target textual format is different from the source file then it is said to be a translation, if it is the same, it is said to be a serialization. If the abstract syntax tree was manipulated before being serialized, either in a reversible or irreversible way, then it is said to be a code transformation.

2.4.2 Code Expansions

Physical equations systems often contain equations that share the structure, that is, they have the same concrete syntax tree. As an example, finite element volume problems are often described by few equation repeated for a large set of variables. Since the quantity of variables is a parameter that may be tuned throughout the development of the analysis, it is natural to expect that a language designed to describe such systems will allow to easily modify such parameters.

A possibility is to allow the user to write multiple equations as a for-cycle. We call for-equation syntax any syntax that is composed of a induction variable i, a induction range r and a equation e that may or may not include a usage of i in its expressions:

$$f(x) = k \qquad \forall x \in X$$

Where X is some set of items $\{x_1, \dots, x_n\}$. Such syntax entails that there exists a equation e_k for each element $x_k \in X$, obtained by replacing x with x_k in the equation $f(x) = k$. Thus, we define as for-equation E the list of elements $e_0, \dots, e_n$, where each $e_x \in E$ is a scalar-equation.

We define vector-variables v as a list $v_1, \dots, v_n$ of variables, for sake of clarity we refer to variables and equations as vector-variables and vector-equations.

In memory representation The for-equation syntax will be held in memory by the compiler in a abstract syntax tree. As an example, it may be structured as the following:

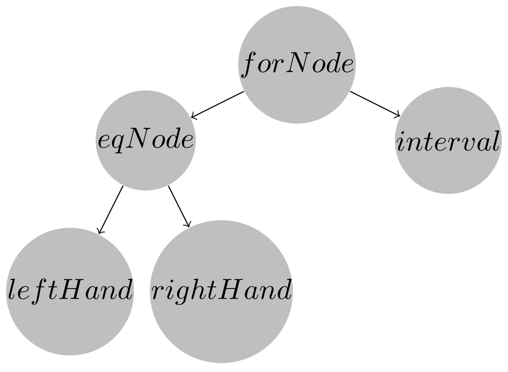


Figure 2.4: Structure of a generic for-equation as represented in an Abstract Syntax Tree

Since a for-equation syntax holds all the information needed to compute the unrolled set of scalar-equation, we would like to preserve such compact representation as long as possible across the compiler stages.

General purpose programming languages implement control flow structure such as the for-loop as a machine code jumps. This implies that they do not require to unroll the for-loop into a list of statements. Not only, it may not even be possible to unroll those loops, since their starting and ending conditions may not be known. The syntax we showed for equations systems is very different. The bounds of the for-loop must be computable at compile time. If they were not computable they would be a unknown variable themselves, and they would contribute to the equations set, rather than specify how it is formed. Furthermore, since the compiler task is to transform a declarative set of equations into a list of instructions, then compiler may need to manipulate the vector-equations as a mathematical object. If that is the case, then transformations of the abstract syntax tree are required. Let us assume that $X(t)$ contained only two variables, then the for-loop would imply only two equations. The root *foorNode* shown in the figure 2.4 can be replaced and the *eqNode* tree can be duplicated. The resulting tree is the one showed in figure 2.5

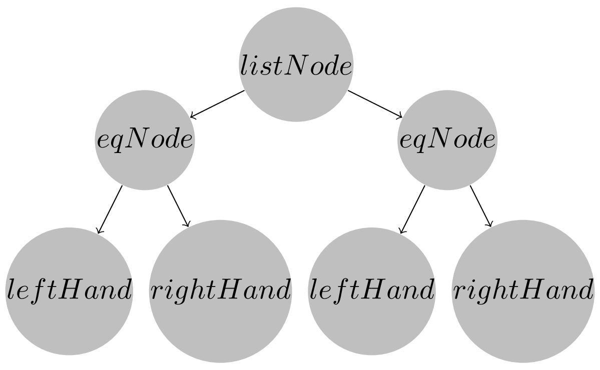


Figure 2.5: Structure of a generic list of equations as represented in an Abstract Syntax Tree. For simplicity, only two equations are shown – in general, any number of *eqNode* istances is allowed.

We call *scalarized-equations* this abstract syntax tree and all equivalent syntax trees .

Notice that if for-equations must be manipulated to produce a simulation, then the task of a compiler of a language describing system of equations is different than simply translating from a high-level description of a program to a low-level one. The abstract syntax tree must be analyzed at compile time to deduce the semantics of the simulation.

Consider again the interval node shown in the previous example, a language may allow to provide an arbitrary expression as bounds of the interval. If such is the case then those intervals must be computed at compile time to be able to perform the unrolling. Thus, not only code expansion is required, but also arbitrary expression computation as well.

2.5 Compilers

There exists multiple projects that can be used as base on which a compiler front-end can be built. Among the most known is *gcc* [4]. *OpenModelicaCompiler* emits

C code that is later compiled by *GCC*. While we may adopt the same strategy, since it is our interest to create the fastest compiler possible we will not do so.

Rather we will focus on the *LLVM* project.

2.5.1 LLVM

The LLVM compiler infrastructure project is a set of compiler and tool-chain technologies [1]. LLVM can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes [2].

The main component of the LLVM project is the LLVM intermediate representation *(LLVM-IR)*. The *LLVM-IR* is a low-level programming language with many similarities to assembly languages. It is a RISC instruction set and most of the platform dependent features are abstracted away. As an example, calling conventions are hidden by the call and ret instructions. There is not a fixed number of registers, rather they are a infinite amount and register allocation is performed in the back-end of the compiler.

The snippet of code in algorithm 7 is a hello world implementation in the LLVM-IR:

Algorithm 7: LLVM example

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

The snippet of code exemplify many important traits of the language:

- the language is typed, and there are no implicit casts.
- Every function declaration has a return type and a type for each parameter.
- Types must be specified at each usage of a variable. This has the advantage of avoiding the need of lookup onto symbols tables.
- The number of parameters is fixed, except for variadic functions, such as printf.
- Signatures of external functions must be specified, since printf is implemented by the c standard library.
- Storage type of symbols can be specified, since .str has no reason to be exposed to other object file at compile time, then it is possible to store it as internal, so that it will not be populate the table of public symbols.

While this is a language that allow fine-grained control on the operations that must be performed at a extremely low-level. It is not optimal describe systems of equations. We will handle such issue in chapter 5.

A note on LLVM-MLIR It must be noted the compiler field is quickly moving to adapt the new *LLVM-MLIR* intermediate representation as core target of various language front-ends. In the long term we believe that would gain much from adopting *LLVM-MLIR* in term of compatibility with other tools and of maintainability. Unfortunately, at the moment *LLVM-MLIR* is not yet mature enough to provide the traits just described.

2.6 Related works

We are aware of only one related work, a method to improve the compilation speed of Modelica has been proposed by Zimmermann, Fernandez and Kofman [23]. They proposed *set-based* graph methods intended to improve the performance of already existing compilers, and their algorithms are extensions of those currently used in *OpenModelicaCompiler*. Their methods suggest to keep the graphs in implicit form and materialize the nodes and edges of the graphs only when the computation cannot carry on in any other way.

Our solutions is similarly based on graphs. The main difference between their solution and ours is that we pose restrictions on the languages we can compile, we execute one of the step of the pipeline in two stages rather than one, and we do not require to modifications of the graphs after they have been built.

2.7 Conclusions

In this chapter we have shown the state of the art in the field of domain specific modeling languages and provided a inside in how their compilers operate. In particular we have shown the minimal pipeline that is not able to be highly performant when applied to the domain of large scale systems. In particular we isolated the problem of array-preservation across the pipeline stages. Furthermore, we have shown *LLVM* and the issues in using *LLVM-IR* as a intermediate representation directly.

Chapter 3

Theoretical Background

I wish my wish would not be granted

Douglas R. Hofstadter

In this chapter we present the fundamental definitions and algorithms that will be used throughout this document. This includes generic definitions regarding graph theory, matching and scheduling algorithms, polyhedral analysis. Furthermore, we will provide simple higher-order function definitions that will be widely used to describe operations on set of indices.

3.1 Graph Theory

The algorithm we discuss in this document are mostly based on graph theory, and in particular they are based on the exploitation of homomorphism between graphs to achieve better performances. Thus, we introduce the definitions of graphs, directed graphs, bipartite graph, graph homomorphism, path, cycles, rank and strongly connected components.

3.1.1 Graphs

A graph G is defined as a ordered pair (V, E) composed by:

- V a set of vertices.
- $E \subset \{\{x, y\} \mid (x, y) \in V \times V\}$ a set of edges, that is, a set of pairs (x, y). There is a edge from vertex x to vertex y iff $(x, y) \in E$.

Directed and undirected graphs If a graph $G = (V, E)$ is such that

$$(x, y) \in E \iff (y, x) \in E$$

then the graph is said to be undirected. If that is not the case it is said to be directed.

Bipartite graphs If a graph $G = (V, E)$ is such that V can be divide into two disjoint subset S_1, S_2 and $S_1 \cup S_2 = V$ and

$$\begin{aligned}(v_1, v_2) \in E &\implies v_2 \notin S_1 \quad \forall v_1 \in S_1, v_2 \in V\\ (v_1, v_2) \in E &\implies v_2 \notin S_2 \quad \forall v_1 \in S_2, v_2 \in V\end{aligned}$$

then the graph G is said to be a bipartite graph. Informally, a graph is said to bipartite if it is possible to divide the verticies into two set such that no vertex in one set is connected to a vertex in the same set.

Paths and Cycles A list $P = [p_1, \ldots, p_n]$ is said to be a path on a graph $G = (V, E)$ if for each adjacent pair of elements $p_x, p_y \in p$ it holds that $(p_x, p_y) \in E$.

A list $P = [p_1, \ldots, p_n]$ is said to be a cycle on a graph $G = (V, E)$ if it is a path on the graph G and $p_1 = p_n$.

We call $path(p, g)$ and $cycle(c, g)$ the function that evaluate to true iff p and c are path and cycle of g respectively.

Graph homomorphism A function f is an *homomorphism* between the graphs G and H iff

$$\{u, v\} \in Edges(G) \implies \{f(u), f(v)\} \in Edges(H)$$

As an example consider graphs 3.1a and 3.1b.

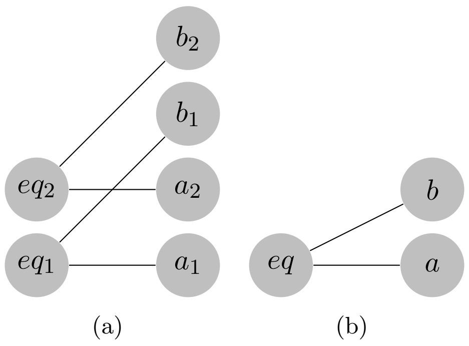


Figure 3.1: Two graphs demonstrating an example of homomorphism.

The function that maps eq_1 and eq_2 in eq, a_1 and a_2 in a, and b_1 and b_2 in b is an homomorphism between graph A and graph B.

Rank We define the function $rank : V \to N$ that maps a vertex with the number of edges containing that vertex.

Strongly Connected Components Given a graph $G = (V, E)$ and a vertex $v \in V$ we call Strongly Connected Component of v in G the set S such that:

$$s \in S \iff \exists P = [v, \ldots, s], N = [s, \ldots, v] : path(P, G) \wedge path(N, G) \qquad \forall s \in V$$

That is, S is the set of vertices that can both reach and are reachable from v.

3.1.2 Bipartite Matching

As we said in Section 2.3.1 conventional Modelica compilers perform the matching and the scheduling steps during compilation. Consider the following snippet of Modelica code, which will be the running example for this section:

```
model Example
 parameter Real x;
 Real y;
 Real z;
 equation
  0 = -x+y; \\eq1
  y = z; \\eq2
end Example
```

The objective of the matching stage is to pair each equation with a single variable, no variable can be paired with two equations. The candidates variables for the matching of an equation are the variables used by the equation itself.

We can model this problem with a bipartite graph $G = (V, E)$. V is composed by the elements $y, z, eq1, eq2$. x does not contribute to the matching because it is a parameter and its value will be known at compile time. E is composed by the elements $(eq1, y), (eq2, y), (eq3, z)$. A graphical representation of G is presented in figure 3.2a.

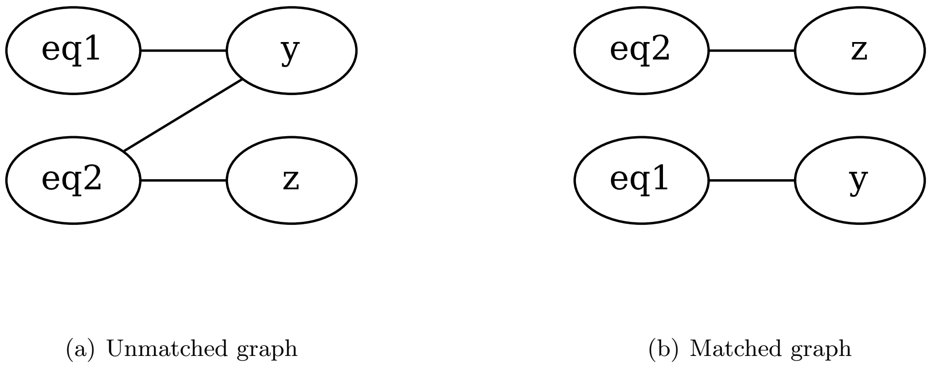


Figure 3.2: Example modeling the relationship between equations (eq1 and eq2) and variables (x, y) before and after matching

When presented in graphical form, the problem of matching consists of selecting a set of edges S such that:

$$\begin{aligned} \exists(x_1, x_2) \in S : x_1 = v \lor x_2 = v \qquad & \forall v \in V \\ distinct(x_1, x_2, x_3, x_4) \qquad & \forall (x_1, x_2), (x_3, x_4) \in S \end{aligned}$$

Where *distinct* is true iff no parameter is equal to another one. Informally, a matching is correct iff each vertex belongs to exactly one matched edge. In the current example there is only one possible matching, that is $S = [(eq1, y), (eq2, z)]$. Such matching is presented in figure 3.2b.

3.1.3 Maximum Flow

A maximum flow problems is a problem of which we have quadruple $Q = (G, s, l, f)$ where:

- $G = (V, E)$ is a graph
- $s \in V$ such that $in(s) = \emptyset$ is called source node
- $l \in V$ such that $out(s) = \emptyset$ is called sink node
- $f : E \to \mathcal{I}$

We wish to find the function $g : E \to \mathcal{I}$ subject to:

$$\begin{aligned} g(e) &\leq f(e) & \forall e \in E \\ \sum_{\forall o \in out(v)} g(o) &= \sum_{\forall i \in in(v)} g(i) & \forall v \in V \setminus \{s, l\} \end{aligned} \tag{3.1}$$

With objective function:

$$max \sum_{\forall i \in in(l)} g(l)$$

Informally, f describes the maximum flow that can travel in a edge, g is the assignment that maximizes the flow reaching the end node that we wish to find. The first constraint states that the flow assigned to a edge cannot be larger than the maximum flow of that edge. The second constraint states that the sum of the flow of all incoming edges of a node must be equal to the sum of the flow of all outgoing edges of that nodes. In other words, there is conservation of the flow.

Residual Graph Consider a maximum flow problem defined by $Q = (Q, s, l, f)$ of which we know a possible solution g that satisfies the constraints. For each Q, g there exists a residual graph $R = (RV, RE)$ composed as such:

$$\begin{aligned} v \in V &\iff v \in RV \\ e \in E \land g(e) \leq f(e) &\iff e \in RE \\ e = (v_1, v_2) \in E \land g(e) \geq 0 &\iff (v_2, v_1) \in RE \end{aligned} \tag{3.2}$$

Informally, such graph has a vertex for each vertex in the original graph. It has all the edge that existed in the original graph iff some flow can still be sent across such edge; we call them forward edges. Finally it has the opposite edge for each edge in the original graph iff any amount of flow can be removed from the edge in the original graph; we call them back edges.

It can be proved that there exists a solution g' such that the constraints are still satisfied and the objective function increases in value iff there exists a path p from s to l in the residual graph. Furthermore, g' can be obtained starting from g and p as follow:

$$g'(e) = \begin{cases} g(e) + 1 & \iff e \in p \land e \in forwardEdges \\ g(e) - 1 & \iff e \in p \land e \in backEdges \\ g(e) & otherwise \end{cases} \tag{3.3}$$

Relationship between Matching and Maximum flow Given a bipartite matching problem, since each vertex of each set must be connected to exactly one node of the other set it is possible to formulate the maximum bipartite matching as a maximum flow problem [22] [20]. This is achieved by connecting each element of one set with a source node, and each element of the second set with a sink node. The maximum capacity of each set of the graph is set to one. The flow graph of the running example is shown in figure 3.3.

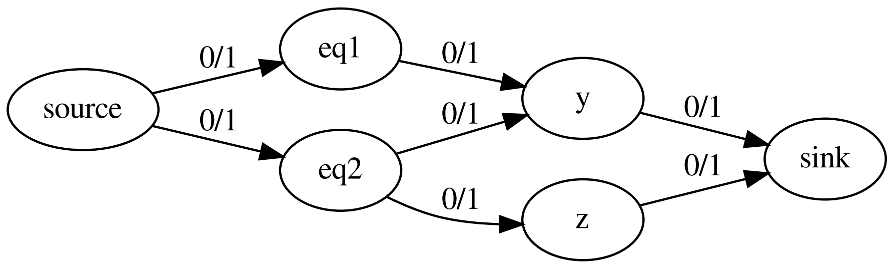


Figure 3.3: Flow graph constructed from Figure 3.2a for solving the matching problem of the running example through finding the maximum flow.

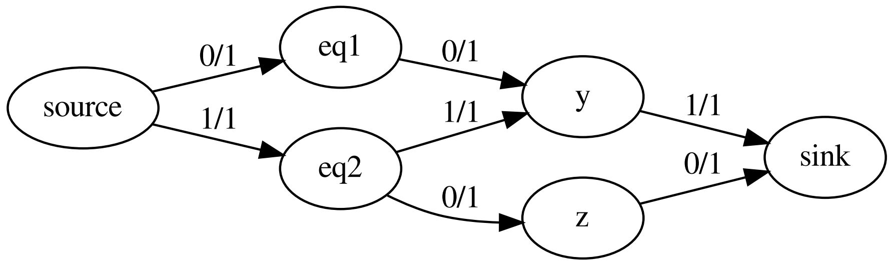
3.1.4 Algorithms for bipartite matching

We now present two bipartite matching algorithm, before doing so we introduce the formal definition of maximum flow problems, since many graph algorithm can be reduced to a maximum flow problem.

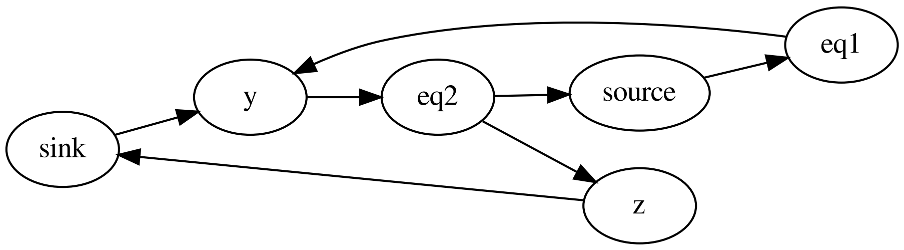
Ford–Fulkerson algorithm Since bipartite matching is a particular case of the maximum-flow problem, to the maximum matching problem can be solved applying the Ford–Fulkerson algorithm [7]. The Ford-Fulkerson algorithm operates by computing residual graph, finding a path from the source to the sink on that graph and then by updating all edges on the path found, Until no more such paths can be found. As an example, we show how the Ford-Fulkerson algorithm matched $eq2$ with y on the previously obtained graph. The first iteration is trivial, the residual graph is identical to the original graph and a depth-first search starting from source will yield a path from source to sink, thus we can increase the flow on those edges, as shown in figure 3.4a.

As the maximum flow of each edge is 1, the residual graph is obtained by inverting every edge with some flow assigned to it. We show the residual graph in Figure 3.4b.

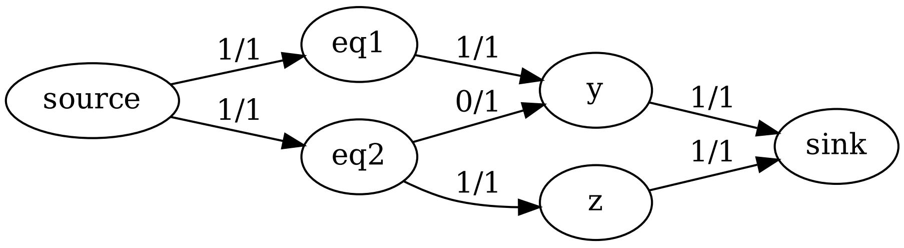
Again, we can improve the flow by finding another path from *source* to *sink*. If no such path is possible, then the current flow is the maximum possible. In this example it can still be improved, due to the path $source \to eq1 \to y \to eq2 \to z \to sink$. Note how $(source, eq1), (eq1, y), (eq2, z), (z, sink)$ all belonged to the



(a) First Iteration



(b) Residual graph after one iteration



(c) Second Iteration

Figure 3.4: Flow graph of Figure 3.3 after the first and the second example iterations of the Ford-Fulkerson algorithm respectively.

$forwardEdges$, $(y, eq2)$ belonged to the $backEdges$ in the residual graph. We can now compute the new assigned flow, which is defined by 3.3. We can see that the total flow is increased.

The graph after the second iteration is shown in Figure 3.4c.

The residual graph now no longer has a path from source to sink, thus the current flow is the maximum possible, and thus the matching is maximum.

The complexity of the Ford-Fulkerson algorithm is $\mathcal{O}(fE)$ where E is the number of edges in the graph and f is the maximum flow of the graph.

Hopcroft–Karp algorithm The Ford-Fulkerson algorithm is able to find the maximum flow on any kind of graph, therefore it is useful to devise a specialized variant of the algorithm specialized for the bipartite matching case. There are various such algorithms, the most well known one is the Hopcroft-Karp algorithm [13].

This algorithm operates in a similar way as the Ford-Fulkerson algorithm. It iteratively builds a matching by executing a depth-first search to build a set of matched edges and by modifying such set at each iteration. After each modification the matched set increases in cardinality, until no path can be found.

The difference lies in the fact that, due to the restrictions on graphs considered, there is no need to build the residual graph, and the only support data structure needed is the set M that will contain the matched edges.

Algorithm 8 shows the pseudo-code of the Hopcroft-Karp algorithm. $findAumentingPath$ is the function that returns the maximum path $P = [p_0, \ldots, p_n]$ such that:

$$\begin{aligned} e &\notin M && \forall e \in edges(p_0) \\ e &\notin M && \forall e \in edges(p_n) \\ (p_x, p_{x+1}) \in M \iff (p_{x+1}, p_{x+2}) &\in M && \forall x \in [0, n-2] \end{aligned} \tag{3.4}$$

Informally the augmenting path is the longest path that alternates between matched and not matched edges. Such path can be found with a depth-first search that does not visits non alternating edges. Thus the complexity is linear with respect to the edge count.

After this path has been found the matching set is updated by removing all edges that were already matched in the path and by inserting those that were not. In order to formally describe this operation, we introduce the set-xor operator $\bigoplus$, that is: the set of elements present in exactly one of both operands. More formally:

$$x \in (L \bigoplus R) \iff (x \in L \iff x \notin R)$$

Algorithm 8: Hopcroft-Karp algorithm

```
Input: $G = (U \cup V, E)$;
Output: $M \subset E$ ;
$M \leftarrow \emptyset$;
$P \leftarrow findAumentingPath(G, M)$;
while $P \neq \emptyset$ : do
    $P \leftarrow findAumentingPath(G, M)$;
    $M = M \bigoplus P$;
end
```

The complexity of this algorithm is $\mathcal{O}(|E|\sqrt{|V|})$. For random graphs the average performance is better, in particular for sparse graph the average depth first search has $\log |V|$ complexity [12]. Thus, the average complexity on sparse graphs is $\mathcal{O}(|E| \log |V|)$.

3.1.5 Scheduling algorithms

As we said in section 2.3.1, the SCC resolution and the scheduling passes require to detect the dependencies among variables. A variable x depends on a variable y if the equation e matched with x in the matching stage requires y to be computed. Thus, it is possible to display the dependencies as a graph $G = (V, E)$. The vertexes V are variables of the systems, and the edges E are the dependencies between equations and variables. Notice that dependency graphs are directed graphs, rather than undirected.

Consider the following already matched snippet of code that will be the running example for the scheduling section.

```
model Example
 Real x;
 Real y;
 Real z;
 equation
  x = 5;
  y = x; \\eq1
  z = y + x; \\eq2
end Example
```

The vertex set V of the dependency graph of this module is equal to $[x, y, z]$ and the edge set E is $[(y, z), (z, y), (z, x)]$ and it is shown in Figure **??**. It is trivial for

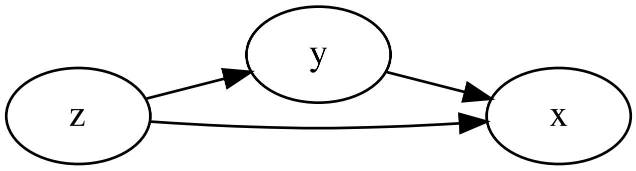


Figure 3.5: Dependency graph corresponding to the example Modelica model.

a human to produce a scheduling that respects the dependencies by looking at the graph. Schedule first every node with outgoing incoming edges, then remove them from the graph all scheduled nodes, then repeat until there are no nodes left.

Strongly connected components and scheduling A graph of dependencies may contain cycles. If it does then it is not possible to find a schedule. Consider the following slightly modified version of the running example.

```
model Example
 Real x;
 Real y;
 Real z;
 equation
  x = 5;
  y = −x + z; \\eq1
  z = y + x; \\eq2
end Example
```

The dependency graph of this model is shown in Figure 3.6. Due to the cyclical

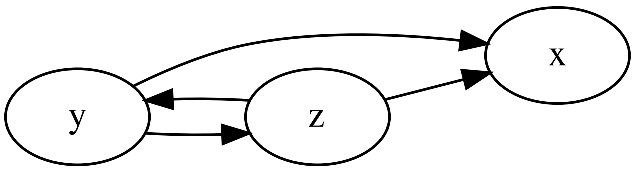


Figure 3.6: Dependency graph corresponding to the second example Modelica model. Note that there is a cycle between variables y and z.

dependency none of $eq1$ and $eq2$ can be scheduled first. From the definition of *SCC* 3.1.1 it follows that every cycle belongs to a *SCC*. Similarly for each *SCC* composed of more than one element in the graph there must be one or more cycles in the graph, and that for any two vertexes in a single *SCC* there must be a cycle involving those two vertexes.

It follows that a graph is schedulable iff there are no *SCC* containing more than one element. Therefore if a Modelica model is provided with circular dependencies, then it must be rewritten is such a way that all *SCCs* are no longer present.

Automatic solvers do exists, unfortunately they are too complex when their input is not constrained to a few equations only. In Modelica implementations is common to calculate approximate solutions that involve at most 3 equations [3]. How such an approximate solver operates is beyond the scope of this work, and will not be discussed in depth, we will rather focus on how to find *SCCs* in the dependency graph.

Khan Algorithm Scheduling is a well-known problem, one of the earliest solutions dates to 1962 and is known as the Khan algorithm [15]. The Khan algorithm operates as a human would, by scheduling the edges from scheduled nodes. The procedure is the following:

Algorithm 9: Khan algorithm

```
Input: $G = (V, E)$;
Output: $L$ ;
$L \leftarrow \emptyset$;
$S \leftarrow [x|x \in V : incomingEdges(x) = \emptyset]$;
while $S \neq \emptyset$ : do
    $s \leftarrow S.extractOne()$;
    $L.append(s)$;
    foreach $n \in successors(s)$ do
        $E.removeEdge((e, n))$;
        if $incomingEdges(n) = \emptyset$ then
            $S.append(n)$;
        end
    end
end
return $L$ ;
```

At each iterations it schedules a node from the schedulable set S and removes all its outgoing edges from the graph. At the same time, if a downstream vertex n is now without any incoming edge, then n is added to S. The complexity of Khan algorithm is $\mathcal{O}(|E| + |V|)$. The main advantage of this algorithm over other scheduling algorithms is that it maintains a global state of all schedulable nodes. Thus, it is possible to use a modified version of it when we wish to schedule nodes in a particular pattern that is not related to the graph's edges. Given a scheduling selection function f we can rewrite the Khan algorithm as follow:

Algorithm 10: Khan algorithm with selection

```
Input: $G = (V, E), f$;
Output: $L$ ;
$L \leftarrow \emptyset$;
$S \leftarrow [x|x \in V : incomingEdges(x) = \emptyset]$;
while $S \neq \emptyset$ : do
    $s \leftarrow f(S)$;
    $L.append(s)$;
    foreach $n \in successors(s)$ do
        $E.removeEdge((e, n))$;
        if $incomingEdges(n) = \emptyset$ then
            $S.append(n)$;
        end
    end
end
return $L$ ;
```

Thus, the khan algorithm is simple algorithm on which to build heuristically a scheduling algorithms.

Tarjan Algorithm The Tarjan algorithm is a well known algorithm to find strongly connected components in a graph [21]. The algorithm operates by executing a depth-first search starting from a random node. If not every node has been reached, it starts a new deep-first search from an unreached node and repeat until all nodes have been visited. Every time a node is visited in one of this searches it is marked with an increasing index, furthermore the starting nodes are marked as as roots of the strongly connected components.

Consider as an example the graph in Figure 3.7. Let us assume that x has

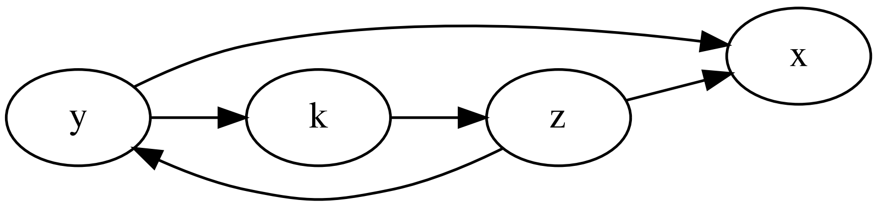


Figure 3.7: Example dependency graph for illustrating the Tarjan algorithm for finding SCCs.

been selected as the starting point of the exploration. None of the other nodes are reachable from x, therefore the search ends immediately. In the second pass, suppose that y has been selected as second root. Starting from y the second search explores the entire graph.

The graph annotated with the tags is shown in Figure 3.8: If we ignore the

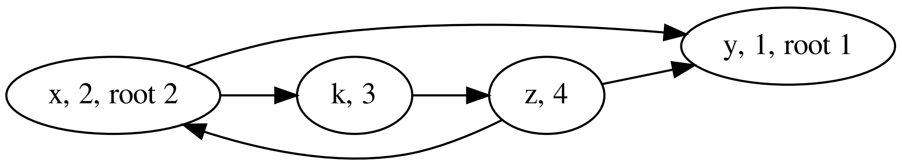


Figure 3.8: Dependency graph in Figure 3.7, as annotated by the depth-first searches of the Tarjan algorithm.

back-edges in the deep-first search - that is, the edges that would have brought the exploration to a already visited node - then we obtain a forest of trees. In our example the forest shown in Figure 3.9

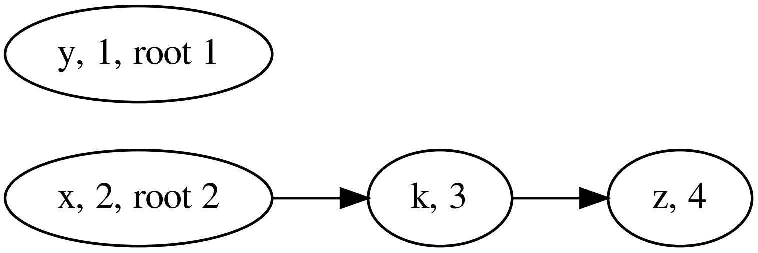


Figure 3.9: Dependency graph in Figure 3.7, as transformed to a forest by the depth-first searches of the Tarjan algorithm. Only the edges that were effectively traversed by the algorithm remain in the graph.

Now we execute another depth-first search on the original graph starting from any leaf v of this forest, in our example we start from element z, with index 4. Nodes that were explored starting from a different root are ignored in this second phase. This can be safely done because if two nodes belong to different roots in the Trajan algorithm forest then either there is no path from the first one to the second or there is no path from the second to the first. If both such paths existed then one would have been found in the first depth-first search of the algorithm.

If in this exploration we encounter a ancestor n of v in the Tarjan forest, that is, a node n such that there exists a path from n to v in the original graph, then both n and v belong to the same *SCC* since we just found a path from v to n as well.

In our example, starting from z, we would visit y, which would be ignored since belong to a different tree. The next node visited is x and thus x, k, z belong to the same *SCC*.

Thus the graph divide in *SCC* is shown in figure 3.10, where nodes are marked with the index of the *SCCs* they belong to.

Tarjan And Scheduling Notice that a depth-first search of the graph is needed to perform the Tarjan algorithm. Thus, as a side-effect the algorithm produce a scheduling of the graph that respect the dependencies. For this reason the Tarjan algorithm is often used in Modelica compiler implementations since it will complete the SCC search and scheduling step at the same time. Unfortunately the Tarjan algorithm does not have the flexibility of Khan algorithm, as it is difficult to integrate with heuristics.

3.2 Polyhedral Analysis

Since we will operate on a set of equations collapsed in for-equation form we must use some kind of representation that allows us to describe the set on indexes used in a for-

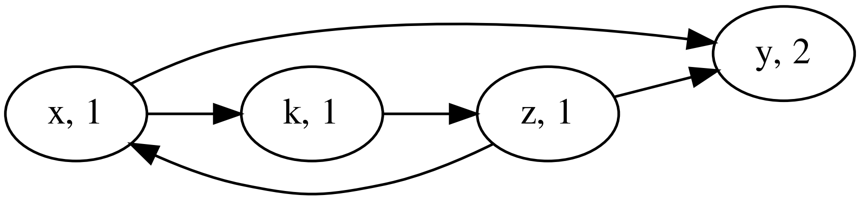


Figure 3.10: Dependency graph in Figure 3.7, as annotated with the final SCCs found by the Tarjan algorithm. The nodes with the same tag identifier are part of the same SCC.

equation. This problem can be easily described as Polyhedral model, that is, a model which treats each loop iteration accessed as lattice points of a polyhedra. Analyses and modifications are performed as affine transformations or more general non-affine transformations generating equivalent polytopes optimized for the particular application. LLVM implements polyhedral analyses in *Polly* [18].

As we will show in Chapter 4, one really useful property that will allows us to reach much lower compilation times is to restrict variable accesses of arrays to invertible functions. The expressive power of polyhedral analysis is much more powerful than we necessitate, thus we will only use a subset of it. Here we show the definitions of integer set to give a intuition of how our solution may be extended and which related works already exists in the field of compilers.

The deviations are equivalent to those presented in Rajopadhye et al. [19]

Basic Integer Set: A basic integer set maps a tuple of integer parameters to a set of integer tuples. As an example consider the following nested for cycles:

```
for (int a = 0; a < k; a++)
  for (int b = a; b <= j; b++)
    c[a][b] = d[a][b];
```

The integer set describing all possibles values of a and b is

$$\mathcal{S}(k,j) = \{(a,b) \in \mathbb{Z}^2 | a \geq 0 \wedge a < k \wedge b \geq a \wedge b \leq j\}$$

That is, a integer set is described as a function from some parameters to the set of all integers that satisfy some expression involving those parameter.

Clearly this function may be arbitrarily complicated, Polly operates on basic integer sets:

A integer basic set is defined as a function $\mathcal{S} : \mathbb{Z}^n \rightarrow 2^{\mathbb{Z}^d} : s \rightarrow \mathcal{S}(s)$ where

$$\mathcal{S}(s) = \{x \in \mathbb{Z}^d | \exists z \in \mathbb{Z}^e : Ax + Bs + Dz + c \geq 0\}$$

with $A \in \mathbb{Z}^{m \times d}, B \in \mathbb{Z}^{m \times n}, D \in \mathbb{Z}^{m \times e}, c \in \mathbb{Z}^m$

We will not delve on the meaning of all parameters of such definition. We only wish to show that there exists a know description of transformation of for-loops and that is based on matrix operations.

Indeed we will show in Chapter 5 that we base our description of for-loops on matrix operations as well.

It must be noticed that the a problem solved by polyhedral analysis is usually solved by exploiting integer linear programming, which is a NP-hard problem. This can be done because integer linear programming is a well optimized field and usually integer linear programming problems handled by polyhedral analysis are small enough to be tractable.

3.3 Higher-order functions

We sometimes refer to $map(s_1, f_1)$ and $fold(s_2, e, f_2)$ functions where s_1 is a set or list of elements belonging to T, $s_2 = [x_1, \ldots, x_n]$ is list of elements belonging to T, e a element belonging to E, $f_1 : T \to E$, $f_2 : E \times T \to E$. Their definitions is:

$$map(s_1, f_1) = \{f_1(x) \mid x \in s_1\}$$

$$fold(s_2, e, f_2) = f_2(\ldots(f_2(f_2(e, x_1), x_2), x_n))$$

Informally, map is the function that given a container and a function it applies that function to each element of the container. fold is a shorthand to the following snippet of code

```
temp = e
for (x : s2)
  temp = f2(temp, s2)
return temp
```

3.4 Conclusions

In this chapter we described the theoretical foundations that we will use in the next two chapters, we show that most of the problems can be described as graph theory problems. This will be useful to prove the complexity of the various stages of the compilation pipeline. Furthermore, we will show in Chapter 5 that our solutions are to some degree modifications of the provided well-known algorithms.

Chapter 4

On the hardness of producing a efficient simulation

The beginning of wisdom is calling things by their proper name.

Mistranslation of Confucius quote

In this chapter we provide the theoretical formulation of the problems we will handle in this thesis as well as proofs of their complexity.

The main difference between the problem formulated in this chapter and those already available in the state-of-the-art is that we do not wish to scalarize all equations ad variables and handled them as unrelated objects, rather we wish to preserve as many for-equations across the pipeline. This in necessary because if it was possible to design all the compilation steps such that they operates in linear time with respect to the number of for-equations but a step yielded scalar variables as an output, then all step downstream would be forced to operate in linear time with respect to the number of scalar variables.

In particular, we will prove that the matching step and the scheduling step have NP-hard complexity. Furthermore we conjecture that the step of SCC search cannot be solved in less than linear time complexity with respect to the number of scalar variables.

4.1 Matching

In the matching stage, the compiler associates each scalar-variable with a single scalar-equation. All variables are associated with a different equation and vice-versa. In other words, it finds a bijection between variables and equations. This well-known task is called bipartite matching. Later stages of the pipeline may benefit from having access to for-equations as well, therefore we introduce additional constraints to the problem in order to maximize the number of for-equations preserved in the output of this stage. We prove that adding this requirement makes the problem NP-hard.

4.1.1 Problem Formulation

The variable access functions are arbitrary, so without loss of generality we will consider a simplified single variable, single dimension problem, where all the vector access of all the equations are in the form $i+k$. More formally given a single vector-variable $V = [v_1, \dots, v_n]$ composed of scalar-variables, and a list of vector-equations $E = [e_1, \dots, e_n]$, each vector-equation is composed of scalar-equations $e_{x1}, \dots, e_{xm}$.

We want to find the bipartite matching that maximizes the number of consecutive scalar equations belonging to the same vector-equation that are matched to consecutive scalar variables equations. We call this problem for-aware matching.

We formalize the for-aware matching on graphs by considering all scalar-variables as one of the vertex set of a bipartite graph, and all scalar-equations as the member of the other set. Therefore, the inputs of the problem will be:

- a vector V of vertexes
- a vector $E = \{e_1, \dots, e_n\}$, $e_x = \{e_{x1}, \dots, e_{xm}\} \forall e_x \in E$ composed by vertexes.
- a set F of pairs (x, y) where $x \in e_k, e_k \in E$, $y \in V$. F contains a pair iff the first member of the pair can be matched with the second.

Informally, each element of V is the scalar-variable, each element of E is a vector-equation, F determines if a scalar-variable was used by a scalar-equation. We define SE as the vector of all scalar equations.

$$SE = \bigoplus_{i=0}^{|E|} E_i$$

where $\bigoplus$ is the concatenation operator.

We want to find the set S of matched edges. The objective function is the number of times in which adjacent elements of V are matched with adjacent elements of the SE. In other words, the number of times consecutive scalar-equations are matched with consecutive scalar-variables.

$$\sum_{Eq \in E} \sum_{e \in Eq} (S(e+1) - S(e) == 1)$$

The only constraint is that every e in S is in the possible matches set F.

$$e \in S \implies e \in F$$

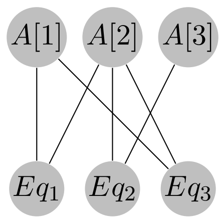
As an example, consider the following system of equations.

$$E = \begin{cases} A[i] = A[i+1] & \forall i \in [1, 2] \\ A[1] = A[2] * 3 \end{cases}$$

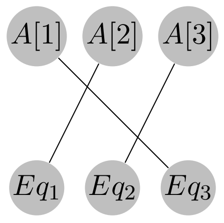
SE is the concatenation of all member of E so it would be a list of 3 elements.

$$SE = \begin{cases} A[1] = A[2] \\ A[2] = A[3] \\ A[1] = A[2] * 3 \end{cases}$$

V is composed by the variables used in the equations, $V = \{A_1, A_2, A_3\}$
The set F of acceptable matches can be shown in graphical form as:



An acceptable solution to the for-aware matching problem of value 1 is:



4.1.2 Proof of hardness

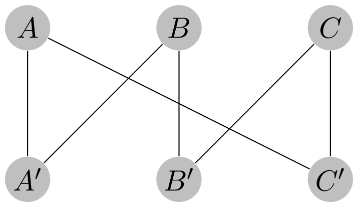
In order to prove that the for-aware matching problem is NP-hard, we reduce the maximum-2-satisfiability problem (max-2-sat) [10] to our problem. max-2-sat calculates the maximum number of clauses that can be simultaneously satisfied in a boolean logic formula in conjunctive normal form. As an example, consider the following conjunctive normal form formula

$$(a \wedge b) \vee (\neg a \wedge \neg c)$$

Max-2-sat maximizes the number of and-clauses evaluated to true. Since a is present in a not expression, it is impossible to find an assignment with value 2, either $(a \wedge b)$ will be true or $(\neg a \wedge \neg c)$ will be. So the maximum value is one, and the acceptable solutions are:

$$\begin{aligned} a = 1, b = 1, c = 1 \\ a = 1, b = 1, c = 0 \\ a = 0, b = 1, c = 0 \\ a = 0, b = 0, c = 0 \end{aligned} \tag{4.1}$$

Max-2-sat is a NP-hard problem, therefore if we can show that every instance of max-2-sat can be solved with the use of the for-aware matching, and that encoding the input data and interpreting the result can be done in polynomial time, then the for-aware matching must be NP-hard. We perform this reduction by encoding the max-2-sat input into the graph, where each variable is represented by a cycle. Before providing the formal algorithm for such construction, we show a simple example of how cycles can be used to achieve the reduction. Consider a cycle C where no vertex $v \in C$ has more than two edges connected to it.



Consider the edge (A, A') in the above example. We know that the matching only matches at most one edge for each variable of the graph. If (A, A') is matched then other edges of A can not. Therefore (A, C') is not matched. Due to the same reason (B, B') must be matched, and so on. By transitivity if the edge (A, A') is matched, then the status of every other edge of the cycle is forced. All the vertical edges must be matched, all the diagonal ones must not.

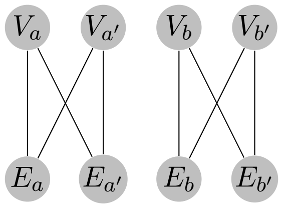
More formally, given a cycle $C = [c1, \ldots, c_n]$ where

$$rank(c) = 2 \qquad \forall c \in C$$

Assuming that c_i is the element of index i mod $|C|$, after the matching the following will hold:

$$(C_i, C_{i+1}) \in S \iff (C_{i+1}, C_{i+2}) \notin S \qquad \forall i \in [0, |C|]$$

Furthermore notice that we can represent a conjunction between two variables by placing two cycles consecutively. As an example, $A \wedge \neg B$ can be encoded as two cycles, one (E_a, V'_a, E'_a, V_a) associated to A and one (E_b, V'_b, E'_b, V_b) to B:



Where (E_a, V_a) is arbitrarily associated with the variable A and the edge (E_b, V_b) is arbitrarily associated with $\neg B$. Under the assumption that $E_{a'}, E_b \in e_k, e_k \in E$ and that e_k is the only element of E with size greater than one. The matching algorithm will try to maximize the number of adjacent matches belonging to the same set. Therefore, it will match the vertical edges of both A and B producing the correct solution for the 2-max-sat problem. The solution is indeed $A = True$ and $B = False$.

Reduction of max-2-sat We provide an algorithm that builds such graphs for every possible input of the max-2-sat problem. Informally, the algorithm proceeds as follow: it scans every and-clause left to right, and for each variable seen it will either start a cycle if that variable has not been seen before, or continue that cycle if it has. It will place two edges of two variables inside the same clause adjacent to each other, so that they will be allowed to contribute to the objective function. After all clauses have been seen, it will close all cycles.

After having built this graph it will invoke the for-aware matching and produce a set of matched edges. From that set, it will deduce the solution of the original problem. Therefore, to perform the reduction we must:

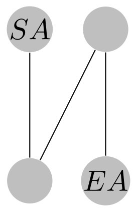
- provide the algorithm
- prove that the encoding and the result interpretation are polynomial
- prove that the result is correct

The algorithm accepts as input a list I of ordered pairs. Each pair is composed of 2 expressions, the expressions are either a equivalent to X or $\neg X$. We assume without loss of generality that the first occurrence of a literal will always be in the positive form. The algorithm returns the dictionary R that maps each literal to its value, true or false. $edges$ is the list of ordered pairs (x, y) equal to F. rmn is the counter used to store the leftmost node reached in the process of building the graph. $vectorEq$ is the set of vector-equations, encoded as a set of indexes i, where i is the index of the two nodes, each belonging to either V or SE. $litCycleStart$ and $litCycleEnd$ are the maps that associate each literal with the index of the vertex that is either the start or the end of the path that is being built from that literal. During the execution of the algorithm we always associate the beginning of a path with indexes from V and the current end of the path with indexes of SE.

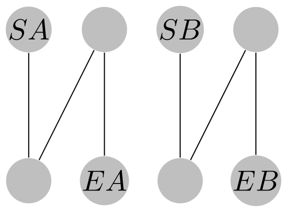
Algorithm 11: Reduction of max-2-sat to the matching algorithm

```
Data: $list(pair(literal,\ literal))\ I$
Result: $map(literal,\ bool)\ R$
$integer\ rmn := 0;$
$map(literal,\ integer)\ litCycleStart := \{\};$
$map(literal,\ integer)\ litCycleEnd := \{\};$
$set(pair(integer,\ integer))\ edges := \{\};$
$list(list(integer))\ vectorEq := \{\};$
$R := \{\};$
for $clause \in I$ do
    for $exp \in clause$ do
        if $exp \notin litCycleStart$ then
            $litCycleStart[exp.lit()] \leftarrow rmn;$
            $edges \leftarrow edges \cup (rmn, rmn);$
            $edges \leftarrow edges \cup (rmn, rmn + 1);$
            $edges \leftarrow edges \cup (rmn + 1, rmn + 1);$
            $litCycleEnd[exp.lit()] \leftarrow rmn + 1;$
            $rmn \leftarrow rmn + 2;$
        else
            if $\neg exp.isNotExp()]$ then
                $edges \leftarrow edges \cup (litCycleEnd[exp.lit()], rmn);$
                $edges \leftarrow edges \cup (rmn, rmn);$
                $litCycleEnd[exp.lit()] \leftarrow rmn;$
                $rmn \leftarrow rmn + 1;$
            else
                $edges \leftarrow edges \cup (litCycleEnd[exp.lit()], rmn);$
                $edges \leftarrow edges \cup (rmn + 1, rmn);$
                $edges \leftarrow edges \cup (rmn + 1, rmn + 1);$
                $edges \leftarrow edges \cup (rmn, rmn + 1);$
                $litCycleEnd[exp.lit()] \leftarrow rmn;$
                $rmn \leftarrow rmn + 2;$
            end
        end
    end
    $vectorEq \leftarrow vectprEq \cup \{\{rmn - 1, rmn - 2\}\};$
end
for $lit \in litCycleStart.keys()$ do
    $edges \leftarrow edges \cup (litCycleEnd[lit], litCycleStart[lit]);$
end
$matchSet := matching(edges, vectorEq);$
for $lit \in litCycleStart.keys()$ do
    $startingEdge := (litCycleStart[lit], litCycleStart[lit]);$
    $R[lit] \leftarrow (startingEdge \in matchSet);$
end
$return\ R;$
```

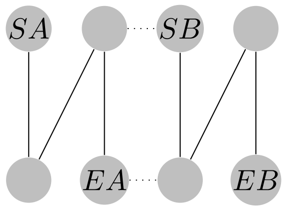
Running example As an example, we show how the algorithm operates on the input $(A \wedge B) \vee (\neg A \wedge C)$. The outer cycle will be executed twice, the inner cycle once. The first iteration of the inner cycle will operate on A. Since A is not present in *litCycleStart*, it is the first time it is encountered, therefore the true branch of the if-statement is executed. Thus, three edges are built, *rmn* is increased by 2, *litCycledEnd* and *litCycleStart* are updated to contain the begin and end indexes of the loop that is being built for A.



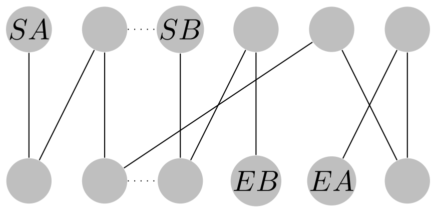
At the second iteration we operate on B and we repeat the same procedure as before, since it is the first time B is encountered as well.



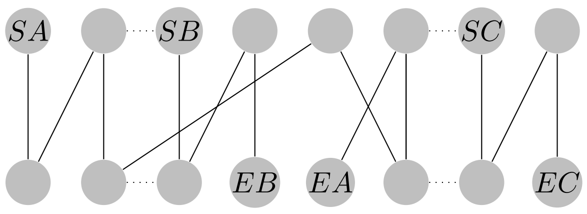
Now we store the information that the two adjacent edges of the edges associated to A and B can contribute to the value:



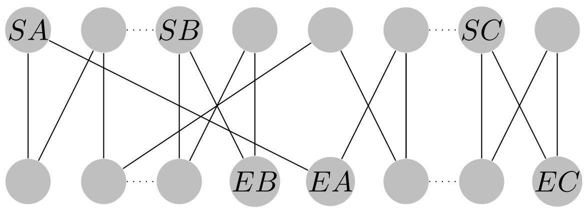
Now the algorithm iterates over a new clause. The first literal is $\neg A$. We have already seen A so we enter the second conditional branch. Since the literal is in negated form, we add 4 edges. *rmn* as well as *litCycleEnd* are updated.



Finally, we place the last literal C exactly as described for the first occurrence of A and B. Then we add the adjacent edges of A and C to *vectorEq*.



At this point the first pass of the algorithm is terminated. Now, we iterate over the collection of seen literals, and we close the cycle in the graph by emplacing the edge from literal end to the literal start.



Now we can invoke the for-aware matching, it can produce value only by matching the vertical edges touching nodes in the dotted regions.

Complexity of interpretation and graph production Constructing the graph requires to iterate over all expressions of the input formula. The body of the loop require constant time. Therefore the graph construction require linear time. The same can be said when interpreting the results.

Proof of correctness To perform the reduction we need to prove that the for-aware matching operating on the graph built by the described algorithm provides the correct solution for each instance of the max-2-sat problem. Or in other words, that the objective function of the for-aware matching is equivalent to the objective function of max-2-sat.

The for-aware matching algorithm can improve the value of the objective function only by matching edges belonging to vector-equations. Since, by construction, every *vectorEq* contains lists of exactly 2 equations then it can only gain value from matching both those equations with the respective scalar-variables with the same index.

Let us call M the set of edges matched by the algorithm, and O the set of edges that are allowed to contribute to the objective function. As stated before we arbitrarily associate the edge of a cycle to the first occurrence of a literal l. Thus, consider a literal l, and the cycle c associated to l. There is path $p \in c$ starting with the edge e allowed to contribute to the scalar objective function, and ending in first edge s in the same cycle. We call $n(x)$ the predicate that determines if a edge x was placed while processing a negated literal.

All the occurrences of the literal in clauses that are not the ones generating the edges e and s always contribute a even number of edges to the path. The occurrence that placed s contributed 3 edges. Since the parity of those contributions is constant, the only contribution that is relevant in order to calculate the parity of the path is the one of e. The quantity of edges contributed by e only depends only on the presence of $\neg$ in the expression l which generated e. Therefore p has odd length iff $n(e)$. Thus:

$$\begin{cases} e \in M \iff s \in M & \mathit{if}\ n(e) = 1 \\ e \in M \iff s \notin M & \mathit{if}\ n(e) = 0 \end{cases}$$

We define $f : E \to L$ the function that maps an edge to the literal associated to the cycle containg it. We define S as the set of all first edges placed in each cycle in C. Consider $f(s)$, as we said the literal associated to $s \in S$ is evaluated to true iff s is the set of matched edges.

$$f(s) \iff s \in M \qquad \forall s \in S$$

From the previous two statements we know that:

$$\begin{cases} f(e) \iff e \in M & \mathit{if}\ n(e) = 1 \\ f(e) \iff e \notin M & \mathit{if}\ n(e) = 0 \end{cases} \quad \forall e \in O$$

We call g the function that associate $e \in E$ to the expression x that was being processed when e has been placed into edges. The $g(e)$ is mapping e to the expression that generated it, $f(e)$is mapping it to the literal inside e and $n(e)$ determines if it was in negated form it follows that:

$$\begin{cases} g(e) \iff f(e) & \mathit{if}\ n(e) = 1 \\ g(e) \iff \neg f(e) & \mathit{if}\ n(e) = 0 \end{cases} \quad \forall e \in O$$

By transitivity:

$$g(e) \iff e \in M \quad \forall e \in O$$

We previously defined the set of vector-equations of the for-aware matching as V. Since for each vector-equation $v \in V$ is composed only by two edges $e_1, e_2 \in O$ we know that the value of the objective function of the algorithm is:

$$\mathit{value} = \sum_{(e_1,e_2) \in V} e_1 \in M \wedge e_2 \in M$$

By substitution with the former statement:

$$\mathit{value} = \sum_{(e_1,e_2) \in V} g(e_1) \wedge g(e_2)$$

We call X the list of $\wedge$ clauses. Since e_1 and e_2 must come from the same $x \in X$ we can replace $g(e_1) \wedge g(e_2)$ with x:

$$\mathit{value} = \sum_{x \in X} x == 1$$

That is: the value that must be maximized is the quantity of boolean clauses that are evaluated as true. However this is the objective of the max-2-sat function. Therefore we successfully performed the required reduction. If we assume that the for-aware matching has polynomial complexity then all steps involved in the algorithm have of polynomial complexity. This is a contradiction since max-2-sat is NP-hard, therefore the matching is NP-hard as well.

4.2 SCC collapsing

After each equation has been matched to a single variable, we shall search all the dependencies among variables. Suppose that variables v_x depends on another variable v_y. If that is the case v_y must be calculated before v_x. If the second variable also depends on the first, then we cannot schedule them, as none of the two can be scheduled first. It is well-known that the dependencies in the scalar case can be found by searching for strongly connected components (SCC) in the graph of dependencies of scalar variables. There are algorithm that can perform this search in linear time. We show how the knowledge of the vector equations allows us to reach better performance than the scalar case. We do this by exploiting the information we can derive from a much smaller vector dependency graph, which is homomorphic to the scalar graph.

4.2.1 Implications of Homomorphisms in the SCC search

Consider the set of matched vector equations $Eq = \{e_1, \ldots, e_n\}$ where e_{ij} is a scalar equation belonging to e_i. Also consider the set of vector variables $Var = \{v_1, \ldots, v_n\}$ where v_{ij} is a scalar variable belonging to v_i. The function $M : Var \to Eq$ returns the scalar equation matched with the scalar variable v. The function $D : Eq \times Var \to \{0, 1\}$ returns true iff the equation has a dependency over the variable.

We define the scalar graph $S = (SV, SE)$ where SV is the set of all scalar variables, and SE is defined such that:

$$(v_{xk}, v_{yj}) \in SE \iff D(M(v_{xk}), v_{yj})$$

That is: there is an edge between scalar variables iff there is a dependency between the scalar equation matched to the scalar variable v_{xk} and the scalar variable v_{jk}.

Additionally, we define the graph $G = (Var, E)$ as follow:

$$(v_x, v_y) \in E \iff (\exists v_{xj}, v_{yk} \implies D(M(v_{xj}), v_{yk}))$$

That is: there is an edge between vector variables v_x and v_y iff there is an edge between the scalar equation that matches scalar variable $v_{xj} \in v_x$, and scalar variable $v_{yk} \in v_y$.

We call f the function that maps a scalar variable to the vector variable it is contained in. f is a homomorphism, since:

$$(v_{xj}, v_{yk}) \in SE \implies (v_x, v_y) \in E$$

We wish to prove that if a variable $v_{xy} \in v_x$ belongs to a cycle C, then $f(v)$ belongs to a cycle as well. C is defined as a list of directed edges $[e_1, \ldots, e_n]$ where the source of the first edge is the sink of the last edge. From the definition of homomorphism we know that:

$$(v_{xj}, v_{yk}) \in SE \implies (f(v_x), f(v_y)) \in E$$

By negating this proposition, we find the following:

$$(f(v_x), f(v_y)) \notin E \implies (v_{xj}, v_{yk}) \notin SE \qquad \forall j, k \in SE$$

That is: if in the graph G there is no cycle between vector variable v_x and vector variable v_y, then there are no cycles containing scalar variables v_{xj} and v_{yk}.

This implies that we can deduce the absence of dependencies among scalar variables belonging to different vector variables just by searching for SCC in the vector graph G. This allows us to perform the dependency search in parallel across different sections of the SCC graph.

Hardness The search of SCCs returns the set containing all SCCs in the graph. This means that it does not need to produce the best possible output for the later stages of the pipeline, since there is only a single correct output possible. Therefore in the worst case it is linear. This is the only algorithm among the 3 analyzed by this document that is not NP-hard.

4.2.2 Conjecture of minimum complexity

Consider the following vector equation, where v is an arbitrary vector variable, and f is a procedure we failed to recognize.

$$v[i] = v[f(i)] \qquad \forall i \in [0, size(v)]$$

There is a cyclical dependency including an element of v iff by applying multiple times f to the same index i we get the starting i as result. In other words, there is a cyclic dependency iff the closure of an element of v under f is a subset v:

$$hasDep(v, f) \iff \exists v_0 \in v : (closure(v_0, f) \subset v)$$

Where $closure(v, f)$ is the smallest set closed under f containing v.

We conjecture that, if there is no closed subset of v, then every implementation of *closure* must invoke the unrecognized procedure f at least $|v|$ times. This because we have to at least check that no element is mapped to itself.

If this conjecture is true then we can always provide a input procedure f to *hasDep* that requires $|v|$ invocations of f, unless f is recognized. Suppose this was not true, then we would have a implementation of *hasDep* that require less than $|v|$ invocations of a not recognized f to find dependencies. If such implementation existed then we can use *hasDep* to implement *closure* and detect if there is no closed subset in less than $|v|$ invocations, but this is a contradiction.

4.3 Scheduling

Once all cyclic dependencies have been resolved, we must schedule the graph. In the scalar case, we schedule all equations without dependencies, then we schedule the equations that depend only on those already scheduled. We repeat until we have scheduled all of them. As we said scheduling a dag, where vertex are the elements to be scheduled and edges precedence that must be respected, can be performed in linear time. We add the objective of minimizing the number of times that scalar-equations belonging to the same vector-equations are not scheduled contiguously.

4.3.1 Problem formulation

This problem can be formalized as follows: given a list of tasks V and dependencies among the tasks described by the set of ordered pairs E, find the schedule S defined as:

$$S(x) \colon V \to [0, |V|], bijective$$

Subject to:

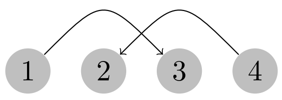
$$S(x) > S(y) \qquad \forall x, y \in E$$

With objective function:

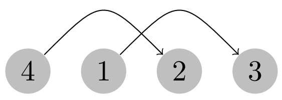
$$\max \sum_{x \in V} (S(x+1) - S(x) == 1)$$

That is: maximize the number of adjacent tasks scheduled in such a way that the distance in scheduling order is exactly one, in other words, that are scheduled one right after the other.

As an example, the following diagram represents the list of vertexes as a sequence of numbered nodes, and the dependency graph as arrows connecting the edges.



One possible schedule is:



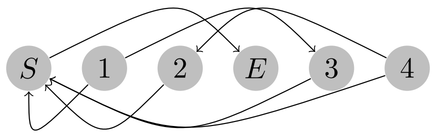
The provided schedule is the best possible one, since only task 4 is scheduled out of order. The score of this schedule is equal to 2: One point derives from scheduling node 1 and node 2 in order. Another point from to node 2 and node 3 scheduled in order as well.

We prove that the scheduling problem is NP-hard by reducing the Graph-To-Dag (GTD) problem to this one. The GTD algorithm creates a acyclic graph by removing the least number of nodes. This problem is np-hard. [11]

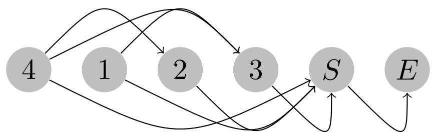
To perform such reduction we describe how to encode an arbitrary directed graph into an acyclic graph, and how to exploit the value function to calculate the minimum number of edges to remove.

Creating Isolated Blocks In our reduction we require to build isolated structures so that their relative position does not influence the objective function. In other words we need the ability of preventing 2 vertexes from being scheduled one after the other. We achieve this by adding one new element E in an arbitrary location and one named S at the beginning. For every previous existing node add the dependency $node \to start$. This implies that the new starting task cannot be scheduled until every other task has been completed. Then add the dependency $S \to E$.This implies that E will never be scheduled before the one to his right or after the one to his left.

Consider the previous example, we can prevent node 2 and node 3 from generating value by inserting S and E among them, and add connections as indicated.

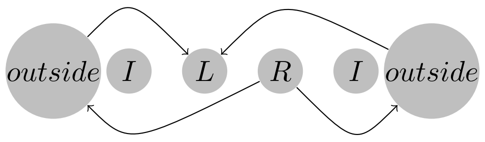


Nodes 2 and 3 of the new graph cannot be scheduled at the same time, therefore the graph would become:



Notice that there is a edge case, if the graph is empty we cannot isolate nodes this way. This is not important because we will not operate on empty graphs. Notice as well that it is necessary for the the edge S to be inserted as first element and the be forced to be scheduled last, so that node 1 cannot be scheduled right after S and thus cannot modify the algorithm score.

Representing nodes Now consider the following list of 4 vertexes in a graph where the I nodes are implicitly isolated as described in the previous section.

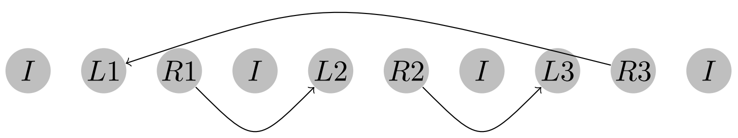


Additionally, we constrain L to only receive arcs from other nodes, R to only emit arcs to other nodes, ignoring the arcs introduced by the isolation process. We call ILRI a list of nodes with these properties. We will use this ILRI nodes to encode arbitrary directed graph, we will illustrate how it can be used to solve the GTD problem for any possible input graph.

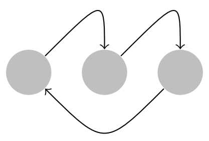
Let us consider a generic graph G composed only of I ILRIes. We call collapsed graph M of the graph G, the graph M that contains a vertex for each element of I. For all edges in G going from a R node to a L node there is edge in the collapsed graph going to the vertex associated with the ILRI of R and the ILRI of L.

If there is a cycle in G then at least one R node cannot be scheduled right after the L node in the same ILRI. If such schedule was possible it would imply that $S(L) = S(R) + 1$. As previously stated, $S(R) < S(L')$ if L' is in the dependencies of R. Therefore, by following the cycle we would obtain $S(L) = S(R)+1 < \cdots < S(L)$. This is a paradox, thus there must be an element scheduled out of order.

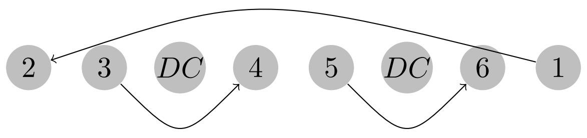
For example consider the following graph:



The equivalent collapsed graph is:



In this scenario L1 is scheduled right before R1, and the same holds for L2 and R2. Therefore $L1 = R1 - 1 < L2 = R2 - 1 < R3$. However, $L3 < R1$, therefore $L3 \neq R3 - 1$.



4.3.2 Proving Hardness

Given the input graph of the GTD algorithm, we call that graph $G = (V, E)$, we build another graph $M = (V', E')$ in which for each $v \in V$ there is a set of nodes that describe an ILRI. We call $I = \{i_1, \dots i_n\}$ the set of such ILRIes. We call $R(i)$ the R node of ILRI i, and $L(i)$ the L node of ILRI i. Since there is a bijection f between V and I that maps a vertex of v to the ILRI associated to it, we abuse the notation defined before hand and allow the use of $R(v)$ and $L(v)$ to indicate $R(f(v))$ and $L(f(v))$ respectively.

We create the set of edges E' of M as follow:

$$(x, y) \in E \iff (R(x), L(y)) \in E' \qquad \forall x, y \in V$$

In other words, there is a directed edge connecting two nodes in the input graph iff there is a directed edge connecting the R node and L node in the destination graph.. Now let us consider the output of the scheduling algorithm when applied on a graph built in such a way. As previously discussed, the algorithm will produce a function $S : V' \to N$ that maps a node of V' to the point in time in which it can be scheduled.

Let us define O as the set of ILRI that had R and L nodes that could not be scheduled one after the other. T is the set of ILRI nodes that could be scheduled one after the other.

$$O = \{i \in I : S(R(i)) \neq S(L(i)) + 1\}$$

$$T = I \setminus O$$

By nature of a graph solely composed by ILRI, the objective function can be gain value only by scheduling the L and R consecutively in the graph. This is consequence of the fact every other node is isolated by construction. As a result we have that

$Value = |T|$ and that $|T|$ is maximized when $|O|$ is minimized. Since every element I correspond to a element of V, then there a bijection f from I to V. We define

$$TV = \{f^{-1}(x) \mid \forall x \in T\}$$

as the set of all elements that generated ILRIES which L and R elements were scheduled one after the other.

Proving output is a DAG Now we wish to prove that $GT = (TV, E)$ is a directed acyclic graph. Let us suppose that GT was not acyclic, then there would be a cycle $C = \{p_0, \ldots, p_n, p_0\}$ such that $C \subset TV$ Since $p_n \in TV$, this means that the L node of the ILRI associated to p_n has been scheduled right before R. Therefore

$$S(R(p_n)) = S(L(p_n)) + 1$$

By construction every edge of the input graph generated an edge between to ILRI structures:

$$(p_x, p_{x+1}) \in E \iff (R(p_x), L(p_{x+1})) \in E'$$

Since the scheduling algorithm is forced to respect the dependencies described by such edge, we have that:

$$(p_x, p_{x+1}) \in E \implies S(R(p_x)) < S(L(p_{x+1}))$$

By combining these results for each element of the cycle C, we obtain:

$$S(L(p_1)) + 1 = S(R(p_1)) < S(L(p_2)) + 1 = ... = S(R(p_n)) < S(L(p_1))$$

However this is a contradiction. Therefore (TV, E') is acyclic.

Proving the amount of removed edges is minimal We wish to prove that every other selection of T with larger cardinality would have generated a cyclic graph.

Let us suppose that given $T' \in I$ with $|T'| > |T|$, and $TV' = \{f(x) | \forall x \in T'\}$ the graph $GT' = (TV', E')$ is acyclic. Remember that we can find a schedule over a DAG in linear time. Let us call the schedule of the GT' graph as $d(TV') \to I$.

Since GT' is acyclic, such schedule is guaranteed to exists. From that schedule we can derive an acceptable solution for our scheduling problem over G'. More formally, consider the schedule $S'(V') \to I$ respecting:

$$s'(v) = \begin{cases} 2d(f(v)) & if\ f(v) \in TV' \wedge v \in L \\ 2d(f(v)) + 1 & if\ f(v) \in TV' \wedge v \in R \end{cases}$$

This schedule assigns consecutive values to L and R nodes of ILRIes that are in T'. As $d()$ respects the optimal schedule on the input graph, then, by construction, L and R nodes of the ILRIes are never scheduled before than L and R nodes of other ILRIes they depend on.

This schedule would have a value of at least $|T'|$, since we created $|T'|$ ILRIes scheduled at the same times. $|T'|$ is greater than $|T|$ and it would respect all constraints. But our scheduling algorithm ensured that $|T|$ was maximum, thus we reached a contradiction.

Therefore we proved that O is the minimum set of nodes that must be removed from V' to create a directed acyclic graph from the graph (V', E')

We call (T, E) the DAG obtained removing the smallest set O from our input graph (V,E). Producing the graph (V', E') and interpreting the output can be both performed in polynomial time. As a result we have reduced the GTD problem to our original problem, which implies it is NP-hard.

4.3.3 Exploiting the graph homomorphisms

Similarly to the previously discussed SCC search, if there is no cycle involving vector-variable A and the vector-variable B then there cannot be a dependency from a scalar variable in B and a scalar variable in A. Therefore, all variables in B can be scheduled after the variables in A.

4.4 Conclusions

In this chapter we proved that two of the three step of the pipeline we considered are NP-hard problems, and we conjectured that the third step cannot be performed faster than linear time. This will deeply effect the design of our solution, since we will be forced to restrict the possible inputs of the pipeline. In other words, we will pose restrictions on the input language so that we can compile it efficiently.

Chapter 5

Design and implementation of MARCO

Cheatham's amendment of Conway's Law: If a group of N persons implements a [COBOL] compiler, there will be N-1 passes. Someone in the group has to be the manager.

Tom Cheatham

As we have shown in chapter 4 compiling the Modelica unrestricted language becomes a NP-hard problem when we add the objective of preserving for-equations and variables arrays across the pipeline. Thus we must select a subset of the language that is expressive enough to be useful.

Furthermore, Openmodelica, the only open-source compiler for the Modelica language, is too complex to be easily changed to match our needs, since it includes many features that are orthogonal to the objective of our work. Thus, we created a new compiler based on the LLVM framework that does not share source code with OpenModelicaCompiler. We have given this compiler the name of M.A.R.C.O. (MARCO), as it is the case for other *LLVM-based*projects, MARCO is a acronym without a meaning.

5.1 High-Level design of MARCO

Since Marco is based on LLVM we have tried to maintain it's design, thus the high level features of Marco are:

- A intermediate representation meant to describe system of equations called *MARCO-IR*. *MARCO-IR* is strongly typed, where all types for each expression must be always provided by the user.
- A in memory and a human readable representations of such *MARCO-IR*.
- Analyses and transformations that transform a system of equations into a executable series of statements.

- A lowerer that translates a valid *MARCO-IR* module into LLVM-IR.
- A Modelica front-end for a subset of the language. Such front-end is completely untied from *MARCO-IR*. It is composed of a parser and a in memory representation.
- A lowerer from the front-end to *MARCO-IR*.

More generally we envision MARCO to be a set of reusable libraries that allow fast transformations of system of equations, can target LLVM as a back-end and provide are more general purpose than just being a Modelica compiler.

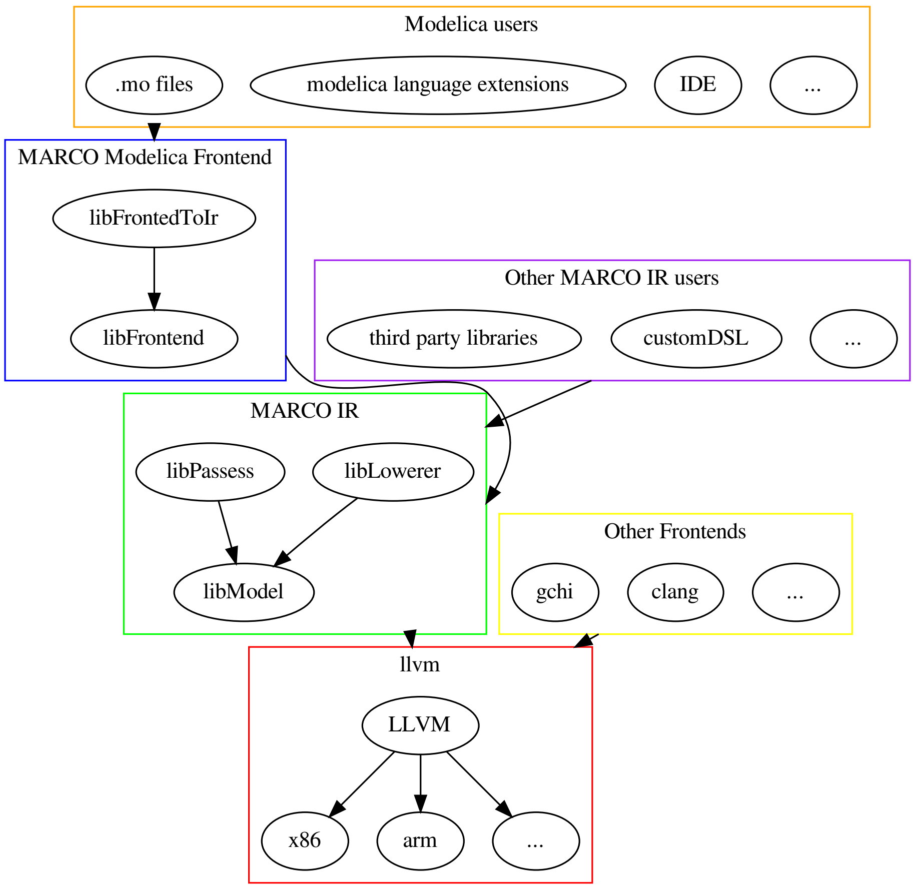


Figure 5.1: The design of the MARCO compiler, represented as sets of interconnected high-level components implemented by reusable libraries.

By dividing MARCO into multiple libraries and tools we allow the users of the system to depend on the minimal set of features that they require to achieve their task.

Furthermore, by decoupling the Modelica language from *MARCO-IR* and *MARCO-IR* from the LLVM-IR we greatly reduce the amount of domain specific knowledge required to use and modify MARCO. As is the case that many compiler engineers are not well versed in physical models so is the case that physical modelers are not well versed in compilers, thus by hiding LLVM in libLowerer, equations transformations inside MARCO-IR, Modelica specific feature in libFrontend we can help users to get acclimatized to the code base.

Finally we wish to allow the user to inspect the standard pipeline, as well as insert its own custom passes and modifications.

A note on MARCO and OpenModelicaCompiler At the current stage Marco does not handle multi-class input Modelica programs, thus we are using open Modelica compiler to translate a generic input .mo file into a flattened model that we can parse.

5.2 The Standard Pipeline

Currently, the standard pipeline by MARCO is shown in figure 5.2 .

The first three steps are specific to Modelica and thus are the least customizable, Modelica abstract syntax tree can be dumped, modified in memory, constructed in memory and parsed from a buffer in memory. However, we do not allow to modify the grammar or the type-checking rules.

Once Modelica AST is lowered to the MARCO-IR we executes the stages of matching, sccResolution, scheduling and EuleroForward. We will describe those steps more in depth later, what we care to present at this stage of the design is that each of those stages accepts as input MARCO-IR and yields as an output MARCO-IR, and such MARCO-IR can be serialized to file. This entails that a user could insert its own modification to the pipeline by simply dumping the IR after a stage, apply its own modification in any way of its choosing and then resume the pipeline at the stage it was left without ever having to touch the MARCO code base.

Similarly, once the model has been lowered into LLVM-IR form then the users are not forced to produce a executable right away. The model can be turned into a library, instead of an executable. It can be instrumented and can be changed for any purposes, again without the need of examining MARCO code base.

5.3 Language Subset

As we proved in chapter 4, the task of preserving arrays across the Modelica pipeline is *NP-hard*, thus we must select a subset of the language and show that for that subset we can execute compilation pipeline in constant time for a given high-level structure.

We consider true the conjecture of minimum complexity shown in the previous chapter. Thus, Marco can have constant time compilation for a fixed high-level structure only if we can always recognize a vector access pattern. In other words we can have constant time compilation for for all models with a particular high-level structure only if, given i a the set of induction variables of a for-equation, any

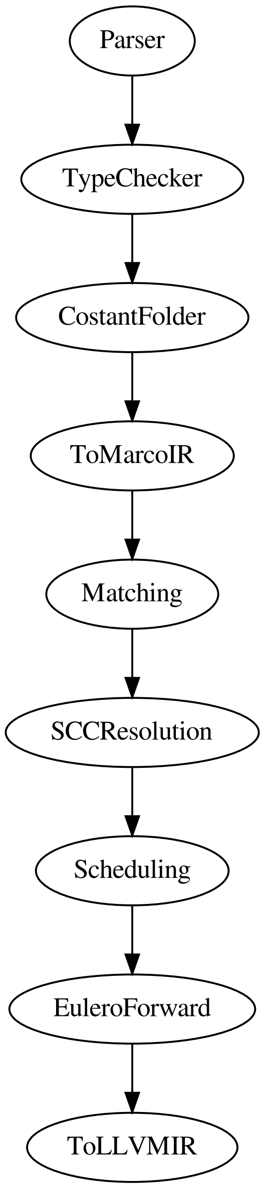


Figure 5.2: Design of the MARCO compiler's pipeline. Note how only the components directly involved in the processing of the Modelica language are represented. The translation from LLVM-IR to machine code is performed by LLVM.

Modelica expression of the kind:

$$vector[f(i)]$$

must be such that f must have been considered when implementing MARCO. We say that a function f that appears as argument of a vector access operation is a vector access function; Thus f cannot be user defined function. Furthermore, since we operates on set of indices s rather than one at the time, we wish that the following operations can be executed in constant time with respect to the size of vectors:

- $map(s, f)$
  We impose this requirement because it often happens that given a vector-equations E iterating on some inductions indices i and involving a vector variable V, we wish compute the set of the indices of the scalar variables belonging to V that are used by any scalar equation belonging to E. If we can extract from E the function g that given a index of a scalar equation e belonging to E returns the index of the scalar variable belonging V that is used by e, then $map(i, g)$ returns the set of indices of scalar variables belonging to V used by any member of E. Thus, if the language poses restriction on both s and f we can compute a really common operation in constant time regardless of the size of V and E.

- function composition $\circ$ defined over the set of vector access functions.
  As we conjectured in chapter 4, the scc-fusion step cannot be performed in less than linear time unless the vector-access functions are recognized. Since we must impose a constraint on those, we may as well use this opportunity to enable other optimizations. Given a vector equation E with a usage of a vector-variables V_1, V_2 accessed with each a vector-access function f_1, f_2 and inductions ranges I. Given the index $i_1 \in map(I, f_1)$, that is the index of a scalar variable $v \in V_1$, we sometime wish to calculate which is is the index $i_2 \in map(I, f_2)$ such that $f_1^{-1}(i_1) = f_2^{-1}(i_2)$.

  In other words, given a scalar variable used by a scalar equation, we wish to calculate which is the index of some other vector-variable used in the same equation. Thus, if f_1 is invertible it holds that: $i_2 = f_2(f_1^{-1}(i_1))$, that is $f_1^{-1} \circ f_2$.

  If we combine this with the previous constraint we obtain that we can calculate this indexes in bulk. Given I_1 a index set containing indices of scalar variables inside V_1, $map(f_1^{-1} \circ f_2, I_1)$ is the index set containing indices that appear at least in a scalar equation belonging E that have at least one usage of a scalar variable indexed by I_1.

  Being able to perform this computation in constant time with respect to the vector-variables sizes is helpful when performing the transitive dependencies calculation present in the scc-fusion step.

- intersection $\cap$ defined over sets of indices.

- union $\cup$ defined over sets of indices.

As we will see in the next section we will able to only accept axis-aligned roto-translation as vector-access function of vector-variables to be able to meet this requirements. This may look like too strong of a requirement. While this would be true for a general purpose programming language it is not the case for languages such as Modelica. Physical models are usually obtained by a quantization of space, phenomena are usually local, and thus it is reasonable that equations would be constrained to only operate with translations and rotations.

Furthermore, it must be noticed that this requirement can be enforced at compile times and the user can receive error messages that can help him write a suitable model.

5.3.1 Vector Access Functions

Given those constraints we selected a subset of linear transformations since linear transformations can be expressed as matrices and thus they, for a given size, require constant time to be applied at a single element and to be combined.

Thus, we only allow vector access function f such that:

$$f(i) = M * i + k$$

Where M is a matrix composed of only zeros and one and has at most one 1 in each row and column. We call S the set of matrices that have this property. That is, we only allow f to be an axis aligned linear roto-translation. We call F the set of such functions.

Consider:

$$f_1(i) = M_1 \times i + k_1$$

$$f_2(i) = M_2 \times i + k_2$$

We wish to prove that the operation of composition $\circ$ is internal to F, that is we must prove that given

$$f_1 \circ f_2 = M_2 \times (M_1 \times i + k_1) + k_2 = M_3 \times i + K_3$$

it is true that $M_3 = M_2 \times M_1 \in S$

This it trivially true since M_1 and M_2 have at most one 1 on each row and thus the matrix multiplication cannot yield values of $M_3 \notin \{0, 1\}$, and cannot yield more than 1 element different from zero on each row.

Notice that a function $f \in F$, $f(i) = M \times i + K$ can be expressed with a single linear effine matrix:

$$f(i) = \begin{pmatrix} M & K \\ \vec{0} & 1 \end{pmatrix} \times \begin{bmatrix} i \\ 1 \end{bmatrix}$$

Since it can be expressed as a single matrix and then the composition is a matrix multiplication and the execution time is constant.

5.3.2 List representation of vector access function

Consider a vector access function f with the following structure:

$$f(i) = \begin{pmatrix} M_{11} & \dots & M_{1n} & K_1 \\ \dots & \dots & \dots & \dots \\ M_{m1} & \dots & M_{mn} & K_m \\ 0 & 0 & 0 & 1 \end{pmatrix} \times i$$

Since, as we said, there is at most one element equal to one in each row and column in the sub-matrix M, this representation can be compressed into list form.

Consider the list $l = [i_1, \dots, i_m]$ where:

$$i_j = \begin{cases} 0 & if \quad M_{ji} = 0 \quad \forall i \in (1, n) \\ d & if \quad M_{jd} = 1 \end{cases}$$

That is, the list where for each row of the matrix there is a element i equal to the index of the element different from 0 in the matrix M, or 0 if there is not such element.

This representation is as informative as the matrix M. from the length of the list we can deduce the height of the matrix, from the max element contained in the list we can deduce the width, and each element of the list will tell us the location of the only element of a row that may be different from zero.

This entails that the pair (l, K) is the only value required to describe a vector access function.

Composition of the list representation Given $f_1 = (i = [i_1, \dots, i_n], K_1), f_2 = (j = [j_i, \dots, j_m], K_2)$ vector access functions in list form we wish to be able to compute the composition of the two $f_3 = f_1 \circ f_2 = (l = [l_1, \dots, l_o], K_3)$ without being forced to transform f_1, f_2, f_3 into matrices M_1, M_2, M_3.

We notice that $o = n$ since in matrix form $M_3 = M_1 \times M_2$ will have the height as M_1, and as we said the height of matrix is tied to the list length in list representation.

We can calculate the values of K_3 as follow:

$$K_3[y] = \begin{cases} K_1[y] & if \quad i[y] = 0 \\ K_2[i[y]] + K_1[y] & otherwise \end{cases}$$

We can calculate the values of l as follow:

$$l[y] = \begin{cases} 0 & if \quad i[y] = 0 \\ j[i[y]] & otherwise \end{cases}$$

Instead of proving this formally we will provide an example, since the formal proof is long and not of much interest. It can be obtained by performing the matrix multiplications and checking which elements are different from zero at the end of the process. Consider the following values for the matrices M_1 and M_2 of f_1 and f_2.

$$M_1 = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

That is, M_1 is describing an access as:

$$v_0[i][j] = v_1[j + 2][i] \quad \forall i \in (0, p_1), j \in (0, p_2)$$

While M_2 describes an access as:

$$v_1[i][j] = v_2[i][3] \quad \forall i \in (0, p_1), j \in (0, p_2)$$

In list form they are:

$$i = [2, 1]$$

$$j = [1, 0]$$

Vector variables v_0 and v_1 can be thought as views, real data is held in v_2 while v_1 and v_2 simply modify the layout of those values. Let us say that we are interested into removing v_1 and we wish to express v_0 as function of v_2 directly. This is a function composition and the matrix multiplication yields:

$$M_3 = M_1 \times M_2 = \begin{pmatrix} 0 & 0 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

M_3 describe the vector access is:

$$v_3[5][i] \quad \forall i \in (0, p_1), j \in (0, p_2)$$

In list form is:

$$l = [0, 1]$$

Our definition yielded the same result. Informally this works because the two functions f_1, f_2 are roto-translations, f_2 is applied first, rotating the axis and adding a translation. After this is done the f_1 is applied, which applied its own rotation and then the translation. Thus, for each row of the matrix f_1 either:

- ignores the contribution of f_2 when the row of the second rotation sub-matrix is composed of only zeros.
- select a dimension from the matrix M_2 that will become the current row, thus providing the $j[i_y] + K_2[i_y]$ component.

Then, the second translation is added, thus providing the K_{1y} component.

A geometric intuition The previous mathematical description is much more convoluted than the problem actually is, let us say that the content of of v_2 is:

1,1	1,2	1,3	1,4	...
2,1	2,2	2,3	2,4	...
3,1	3,2	3,3	3,4	...
4,1	4,2	4,3	4,4	...
...	...	...	...	...

As we said the roto-translation M_2 is:

$$M_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

Which can be thought as:

- Do not apply any transformation horizontally, since the first row of M_2 is $1, 0, 0$.
- Replace the second value of the pair with 3, since M_2 is $0, 0, 3$.

Thus, V_1 content is:

1,3	1,3	1,3	1,3	...
2,3	2,3	2,3	2,3	...
3,3	3,3	3,3	3,3	...
4,3	4,3	4,3	4,3	...
...	...	...	...	...

Then we must apply transformation M_2 that was:

$$M_1 = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Thus:

- The columns of the input arrays become the row of the output matrix, since there is a 1 in the second element of the first row. They are furthermore translated by 2 to the left.
- The row of the input array become the column of the output matrix. since there is a 1 in the second element of the first row.

Thus the content of V_0 is:

5,1	5,2	5,3	5,4	...
5,1	5,2	5,3	5,4	...
5,1	5,2	5,3	5,4	...
5,1	5,2	5,3	5,4	...
...	...	...	...	...

The relationship between V_0 and V_1 is that each row of the output matrix is equal to the column of the source matrix starting from the location $5, 1$ and moving vertically.

This is indeed the matrix:

$$M_3 = M_1 \times M_2 = \begin{pmatrix} 0 & 0 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Which does exactly the same:

- due to the first row equal to $0, 0, 5$ replace the first member with 5
- due to the second row equal to $1, 0, 0$ each second member is equal to the input first member.

5.3.3 Indexes Sets

If we restrict the possible vector access functions to the set F of axis aligned roto-translations, then the set of indexes arise naturally from what we can express with those functions. Consider an interval $i = (b, e]$:

$$i = \{x \mid x \in \mathbb{N} \wedge b \leq x \leq e - 1\}$$

We call the set I of all possible values of i as the set of mono-dimensional intervals. A list $l = [i_1, \dots, i_n]$ of n mono-dimensional intervals is a multidimensional interval. We abuse the notation and we define $\vec{i} \in l$ as follow:

$$\vec{v} = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} \in l \iff x_k \in i_k \quad \forall k \in (1, n)$$

Furthermore we say that l has n dimensions, or that the dimensionality of l is n. We call L the set of all possible multidimensional interval.

Example Consider the following for-equation:

$$\dot{x}[i] = f(x[i]) \qquad \forall i \in (0, 5]$$

This for-equation describes 5 scalar-equations, each of them operating over a distinct element of the vector x. The mono-dimensional interval equivalent to the induction range of this vector-equation is $i = (0, 5]$, the multidimensional interval equivalent to this vector-equation is the list $l = [i]$.

Consider now what would change if x was instead a multidimensional vector $x : int[][]$, and the previous vector-equation was instead:

$$\dot{x}[i][j] = f(x[i][j]) \qquad \forall i \in (0, 5], j \in (1, 10]$$

Then there would not be a mono-dimensional interval that describe the induction range of such for-equation, since a mono-dimensional interval cannot describe both ranges i and j at the same time.

It is still possible to find a equivalent multidimensional interval that describe the induction set, such set is $l = [(0, 5], (1, 10]]$. Such set l has dimensionality equal to 2.

Min-max representation We presented the multidimensional interval as a list of pairs. Such set may be described as a pair of lists as well. The first list containing the minimal point(min) of the range on each dimension, and the second point(max) containing the first point not inside the set on each dimension. As an example, the multidimensional interval $l = [(0, 5], (1, 10]]$ can be written in min-max representation as

$$l = ([0, 1], [5, 10]]$$

Thus, a n dimensional interval is equivalent to a pair of n dimensional vectors:

$$\vec{b} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \vec{e} = \begin{pmatrix} 5 \\ 10 \end{pmatrix} \quad l = (\vec{b}, \vec{e}]$$

Sum in min-max representation Given two multidimensional interval in min-max form s_1, s_2 we say that they are both contiguous and summable iff

$$s_1 \cap s_2 = \emptyset$$

$$\exists m \in l \quad m = s_1 \cup s_2$$

That is, two vector are summable iff they are disjoint and their union can be represented as multidimensional interval.

Informally, this is true iff s_1 and s_2 are different from each other on a single dimension, and on that dimension their intervals are contiguous.

Example Consider the following multidimensional intervals:

$$m_1 = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 10 \\ 10 \end{pmatrix} \right)$$

$$m_2 = \left(\begin{pmatrix} 0 \\ 10 \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \end{pmatrix} \right)$$

Informally $m1$ is a square 10x10 starting from $(0, 0)$ while $m2$ is as well a square 10x10, but it is located in $(0, 10)$. Since they differ just by the values of one dimension and on that dimension they are contiguous then $m2$ can be seen as an expansion of $m1$, indeed their sum can be represented as:

$$m_1 + m_2 = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \end{pmatrix} \right)$$

Interaction with Vector Access Function Consider now a multidimensional interval in min-max representation $l = (\vec{b}, \vec{e}]$, and a axis aligned roto-translation $f(\vec{i}) = M \times \vec{i} + \vec{k}$ where the height of matrix M and vector $\vec{k}$ is the same as the dimensionality of l. We claim that computing $map(l, f)$ is a constant time operation. This is true because

$$map(l, f) = (f(\vec{b}), f(\vec{e})]$$

that is, to calculate $map(l, f)$ we only need to apply the function f to the min vector $\vec{b}$ and the max vector $\vec{e}$.

This is true because f is the sum of an axis aligned rotation and translation, thus the translation will simply translate the boundaries of the set and the axis-aligned rotation will only only reorder the elements of $\vec{b}$ and $\vec{e}$.

Intersection of multidimensional intervals Given two multidimensional intervals in list form $i_1 = [x_1, \ldots, x_n], i_2 = [y_1, \ldots, y_n]$ we can can calculate $i_1 \cap i_2$ by simply performing the intersection on each dimension. That is:

$$i_1 \cap i_1 = [x_1 \cap y_1, \ldots, x_n \cap y_n]$$

Example Consider the following for-equation:

$$x[\dot{i}][j] = g(x[j][i+3]) \qquad \forall i \in (0,2], j \in (8,10]$$

The multidimensional set equivalent to the induction ranges of such equations is

$$l = (\vec{b}, \vec{e}] \quad \vec{b} = \begin{pmatrix} 0 \\ 8 \end{pmatrix} \quad \vec{e} = \begin{pmatrix} 2 \\ 10 \end{pmatrix}$$

The vector access function f used in the right hand member of the equation to access variable x is the function that given the induction variables $\begin{pmatrix} i \\ j \end{pmatrix}$ returns $\begin{pmatrix} j \\ i+3 \end{pmatrix}$.

Thus

$$f(\vec{i}) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \times \vec{i} + \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

Let us assume now that we are interested in calculating which elements of vector-variable x are accessed by the right hand of the presented equation. This is equivalent to computing $map(l, f)$.

As we said

$$map(l, f) = (f(\vec{b}), f(\vec{e})]$$

. By performing the calculations

$$f(\vec{b}) = \begin{pmatrix} 8 \\ 3 \end{pmatrix} \quad f(\vec{e}) = \begin{pmatrix} 10 \\ 5 \end{pmatrix}$$

Indeed, this is the same result we would have achieved by manually applying f on each element of the multidimensional interval.

5.3.4 Non-contiguous index sets

We shown how we can perform function composition and mapping of index sets in constant time by utilizing multidimensional interval and axis aligned roto-translations. Unfortunately this is not enough to build a useful compiler on top of them. Set operations $\cup$ and $\cap$ are not internal to the set of multidimensional interval.

To be able to express those operations, given the set S of all possible multidimensional index set, we call generalized multidimensional index sets(GMIS) all $G \subset S$ where all elements of G have the same dimensionality. As before we abuse the notation and we define that $\vec{v} \in G$ as follow:

$$\vec{v} = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} \in G \iff \vec{v} \in e \quad \exists e \in G$$

That is, a vector belongs to G if it belongs to any of its contained elements.

Informally, a GMIS is just a list of multidimensional interval . This is helpful, because given two multidimensional interval s_1, s_2 then $s_1 \cup s_2$ is simply equal to $\{s1, s2\}$. Similarly, given two GMIS $g_1 = \{s_{11}, \dots, s_{1n}\}, g_2 = \{s_{21}, \dots, s_{2n}\}$ then $g_1 \cup g_2 = \{s_{11}, \dots, s_{1n}, s_{21}, \dots s_{2n}\}$

This allows to compute $\cup$ in constant time with respect to vector sizes.

GMIS min-size representation: Clearly, given a set of n dimensional vectors there are infinite GMIS describing such set. Still, there exists some representations that have minimal length, or in other words where the number of multidimensional intervals inside a GMIS is minimized.

We say that the minimal GMIS $c = \{e_1, \dots, e_n\}$ equivalent to a given GMIS n is the GMIS such that n is minimum.

$$\vec{v} \in c \iff \vec{v} \in n$$

$$e_x \cap e_y \neq \emptyset \implies x \neq y \quad \forall x, y \in (1, n)$$

That is, c contains exactly the same elements as n, the multidimensional interval that compose c are disjoint, and the number of multidimensional interval required to described it is minimum.

Reaching minimal size Given a GMIS $g_1 = \{e_1, \dots, e_n\}$ it is not trivial to find the minimal GMIS when all elements e_x are not not disjoint but is possible to perform such operation when they are.

If they are disjoint we only require to find an equivalent $g_2 = \{r_1, \dots, r_m\}$ such that the m is minimal.

From the definitions of canonical GMIS and from the assumption of all elements $e_x \in g_1$ being disjoint it follow that we can partition the elements of g_1 into a set of sets $P = \{\{e_{11}, \dots, e_{1i}\}, \dots, \{e_{o1}, \dots, e_{ok}\}\}$ such that

$$\forall r \in g_2 \quad \exists p \in P \quad \bigcup_{p_j \in p} p_j = r$$

Informally, this statement means that the multidimensional intervals composing a GMIS can be cut into smaller intervals and then recomposed to obtain the minimal GMIS.

Since the multidimensional intervals are described by a pair of min and max vectors then the only possible way to compose two multidimensional index set is if they can be summed as shown earlier.

Thus, to reach the minimal GMIS only require us to iterativelly sum all multidimensional index set that can be summed until no two remaining set can be summed.

This implies that set operation $\cup$ can be performed into constant time with respect to vector sizes when implemented to be internal to the set of minimal GMIS.

When we need to calculate the union of two GMIS we simply concatenate the sets and then we merge all multidimensional elements that are summable.

This does not always yields the minimal GMIS since it depends on the order in which they are summed, but it is effective enough for our purposes and has the advantage that even if a sub-optimal merge is operated, it can be later reverted when the missing pieces are inserted and they are merged into a larger multidimensional interval.

Implementing intersection Now we are interested in how implement set operation $\cap$ on canonical GMIS. The operation does not require to modify the underlying data structure.

We simply intersect each element of one GMIS with each element of the other.

Set minus of multidimensional intervals Given two multidimensional intervals in min-max form $i_1 = (b_1, e_2), i_2 = (b_2, e_2)$ it is more complex to calculate the set-minus operation defined as:

$$i_3 = i_1 - i_2 = \{x \mid x \notin i_2 \quad \forall x \in i_1\}$$

If the dimensionality of i_1 and i_2 is one then we would need to only calculate the mono-dimensional interval set-minus operation. Such operation can be performed by splitting the the interval i_1 at the start and end points of i_2, where the operation splitting s a mono-dimensional interval i at into point x is defined as

$$s_{ic}(i, x) = [\{j \mid j < x \quad j \in i\}, \{j \mid j \geq x \quad j \in i\}]$$

That is, the result of a splitting operation is a list of two elements where the first element is the interval that contains the range less than x and the second element is the range greater or equal to x.

We define a overload of s that operates on two mono-dimensional intervals $x = (b_x, e_x), y = (b_y, e_y)$:

$$s_{ii}(x, y) = [\{j \mid j < b_y \quad j \in x\}, \{j \mid j \geq b_y \land j < e_y \quad j \in i\}, \{j \mid j \geq e_y \quad j \in x\}]$$

That is, the list of sets where the first element is the interval of which all points are less than any point in y, the second element is the internal where all points are inside y, and the third element composed of all remaining points.

Given this definition then set minus for mono-dimensional internals x, y is just $s(x, y)$ where the second element has been removed.

We can generalize the definition for multidimensional intervals, given a multidimensional interval i, a cut point x and a dimension n we overload s to be defined as:

$$s_{mc}(i, x, n) = [\{j \mid j[n] < x \quad j \in i\}, \{j \mid j[n] \geq x \quad j \in i\}]$$

That is: the output of the split operation is a list where the first element contains all points where their nth coordinate is less than x and the second element contains all other points.

As before, it is trivial to generalize this definition to operate on mono-dimensional interval instead of a scalar cutting point. Given x multidimensional interval, y mono-dimensional internal, n integer:

$$s_{mi}(x, y, n) = [\{j \mid j[n] < b_y \quad j \in x\}, \{j \mid j[n] \geq b_y \wedge j[n] < e_y \quad j \in i\}, \{j \mid j[n] \geq e_y \quad j \in x\}]$$

Since the output of this operation is a list of multidimensional interval it is equivalent to a GMIS, thus it can be operate on GMIS as well, and in that case both input and output will be GMIS.

If the $x = [x_1, \ldots, x_n]$ parameter is a GMIS, that is a list of multidimensional intervals then s operates by being applied to all its elements.

$$s_{li}([x_1, \ldots, x_n], y, n) = concat(s_{mi}(x_1, y, n), \ldots, s_{mi}(x_n, y, n))$$

Furthermore, if we wish to operate on two dimensions at the same time we can generalize s to operate on an arbitrary number of parameters:

$$s_{li*}(x, y_1, n_1, \ldots, y_k, n_k) = s_{li*}(s_{li}(x, y_1, n_1), y_2, n_2, \ldots, y_k, n_k)$$

That is, splitting can be operated iterativelly, and at each iteration we split all multidimensional intervals obtained at the previous iteration at the new cutting point.

Then, we can generalize this operation to operate on a GMIS x and a multidimensional interval in list form $i = [i_1, \ldots, i_n]$:

$$s_{gl}(x, i) = s(x, i_1, 1, \ldots, i_n, n)$$

That is, for each dimension, we iterativelly split the input GMIS on the mono-dimensional range $i_k \in i$ associated to that dimension.

Finally, we generalize so that split can operate on two GMIS $g_1, g_2 = [m_1, \ldots, m_n]$:

$$s_{gg}(g_1, g_2) = s_{gl}(g_1, [m_1, \ldots, m_n])$$

That is, we iterativelly split g_1 for each multidimensional interval $m \in g_2$.

Informally s accepts two index set and returns the list of multidimensional intervals $l = [i_1, \ldots, i_n]$ such that each element i_k is either fully contained into a element of g_2 or is fully not contained. Thus we can say that:

$$g_1 - g_2 = \{i \mid i \notin g_2 \quad \forall i \in s(g_1, g_2)\}$$

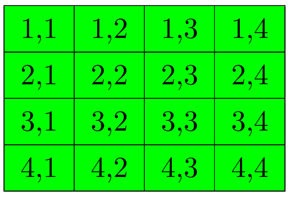
Thus, we have a definition of operation set minus that is not dependent on the size of sets, and thus is able to operate in constant time with respect to array sizes.

An set minus example Let us consider the multidimensional intervals in min max form:

$$\vec{b}_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \vec{e}_1 = \begin{pmatrix} 5 \\ 5 \end{pmatrix} \quad i_1 = (\vec{b}_1, \vec{e}_1]$$

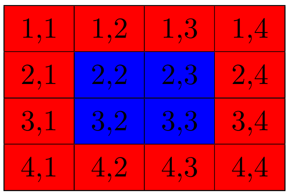
$$\vec{b}_2 = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \quad \vec{e}_2 = \begin{pmatrix} 4 \\ 4 \end{pmatrix} \quad i_2 = (\vec{b}_2, \vec{e}_2]$$

$\vec{i}_1$ can be graphically represented as:



1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

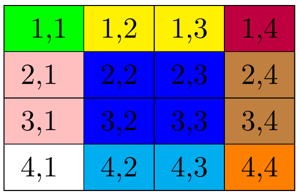
Where elements with color different from red are elements that do belong to the multidimensional interval. $\vec{i}_2$ can be represented as:



1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

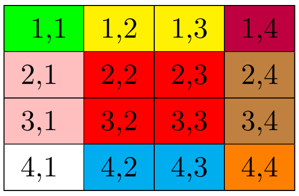
The result $\vec{i}_1 - \vec{i}_2 = \vec{i}_3$ obtained by with the splitting operation is obtained as follow.

- For each dimension split $\vec{i}_1$ at the begin and end point of $\vec{i}_2$



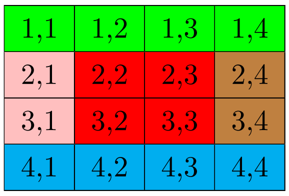
1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

- Remove the spitted element that are fully contained in $\vec{i}_2$



1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

- Merge all elements that can be merged:



1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

Indeed this is a valid representation of $\vec{i}_3$

5.4 Vector-Matching

The first stage of the pipeline requires us to match a scalar-equation with a single scalar-variable. As we said in chapter 3 this a well-known problem, called bipartite matching. As we said in chapter 4, if we add the objective of minimizing the number of adjacent variables that are not matched with adjacent equations the problem becomes NP-hard. Thus, we require an heuristic to be able to perform such matching quickly and to yield a reasonable amount of preserved vector-equations.

Instead of basing our solution on the bipartite matching we rather describe it as a max-flow problem. As an example, consider the following model:

```
model Example
 Real x[2];
 Real y[2];
 equation
 for i in 1:2 loop
  x[i] = 2*y[i]; //EQ1
 end for;
 for i in 3:4 loop
  x[i-2] = 5; //EQ2
 end for;
end Example
```

We call scalarized matching graph the graph normally used to perform the bipartite matching on the flattened model. It this case the scalarized matching graph is shown in figure 5.3: Where arrows represent nodes that can be matched together.

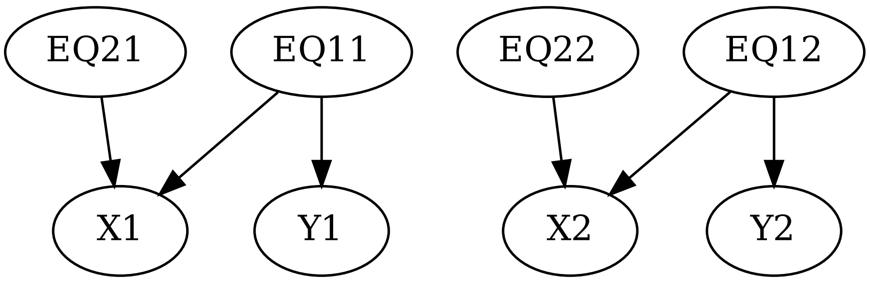


Figure 5.3: Scalarized matching graph of the aforementioned example Modelica code. Note how each equation is represented by multiple nodes to reflect the semantics of the *for* construct: EQ11 and EQ12 for EQ1, EQ21 and EQ22 for EQ2. The vectors x and y are also transformed to scalar variables: X1 and X2 for the two elements in x, and Y1, Y2 for the two elements in y.

We call vector matching graph the graph defined by the homomorphism h that maps all the nodes belonging to the same vector-equation or vector-variable in the scalar matching graph to the same node in the vector matching graph. Informally, the vector matching graph is obtained by merging together all the nodes that were in the same vector in a given Modelica source file.

In this example the vector matching graph is shown in figure 5.4. That is, the

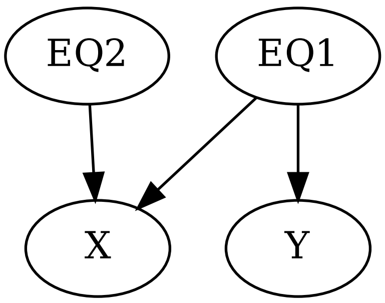


Figure 5.4: Vectorial matching graph of the aforementioned example Modelica code. Note how each equation and vector is represented only once, without expanding the for loops.

vector matching graph of a source model is a graph were there is a vertex for each vector-variable and vector-equation of the source model, and there is an edge between two nodes if any scalar variable mapped to one of those nodes can be matched with any scalar equation mapped with the other node.

We wish to augment such graph by annotating it with the vector-access functions that each equation uses to access each variable, as well as the dimensions of each vector-variable and vector-equation, and the induction ranges.

Such representation is as expressive as the scalar representation and the source file representation, and it is possible to move from any of the three to the other two.

We will provide a heuristic algorithm that uses the information provided by the vector matching graph to perform the matching in linear time with respect to the vector matching graph.

5.4.1 An example of solution

The algorithm we provide is an heuristic one, to explain it we will start with an example and then we will specify the details.

Consider again the model and the graph in the previous example. We look for a matching iterativelly. At first we set all the indexes as not matched. Such state is shown in fig 5.6

Then we select the largest unmatched vector-equation, in this case they are equal thus we pick one in alphabetical order, thus we select $EQ1$. Then we select the variable such that the highest number of scalar equations can be matched with the already selected vector-equation. In this case both X and Y can be selected,

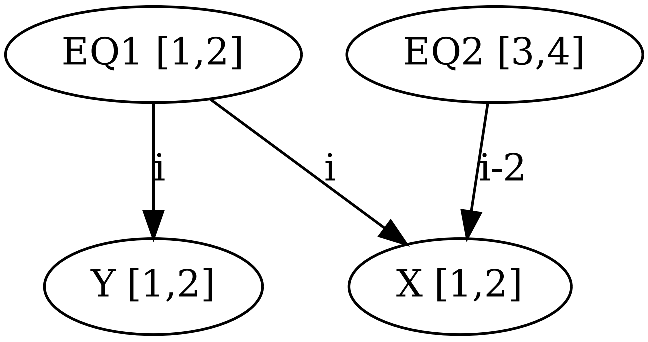


Figure 5.5: Vectorial matching graph from Figure 5.4, annotated with the induction ranges on the equation nodes, the dimensions of each vector variable on the equation nodes, and the vector access functions on the edges.

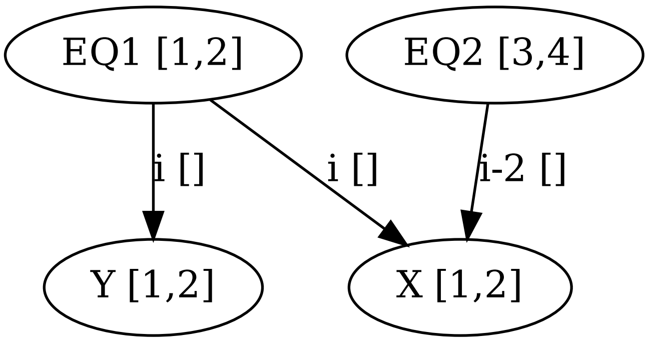
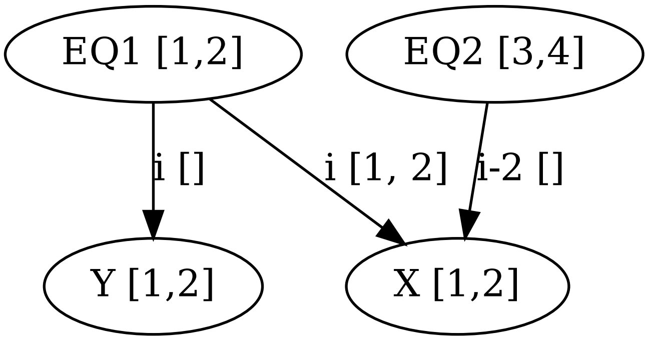


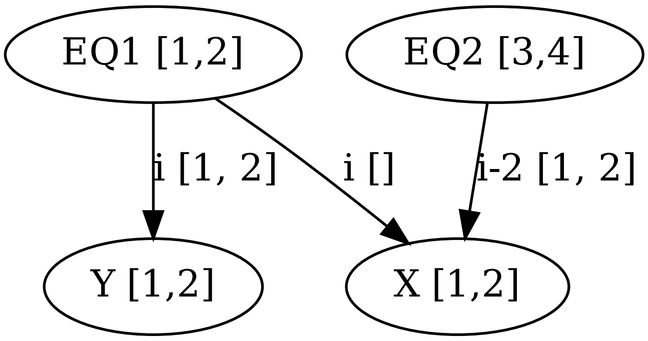
Figure 5.6: Initial state of the vectorial matching algorithm. The vector access functions are annotated with the set of array indices matched to the corresponding equation.

since they can be both matched with $EQ1$ with both their indexes. We prefer X due to alphabetical sorting. We match all the scalar equations of $EQ1$ with all scalar variables of X that can be matched. The resulting matching graph is shown in figure 5.7

Now we iterate again, the next vector-equation with the most unmatched scalar-



(a) 1 iteration



(b) 2 iteration

Figure 5.7: First two iterations of the vectorial matching algorithm starting from Figure 5.6.

equations is $EQ2$. $EQ2$ can only be matched with X, thus we must undo the matching we previously assigned. We operate as follow. We keep track of the list l of currently visited nodes, that is $l = [EQ2, X]$. Then we select the vector-equation that was has the maximum number of elements matched with X that intersect the maximum number of nodes that $EQ2$ may match with X. $EQ2$ can match at most 2 elements with X. $EQ1$ has two elements matched with X and those two elements are the same. Thus we select $EQ1$ for unmatching. We push $EQ1$ at the end of the list l. Now we repeat the matching step process starting from $EQ1$, except that we consider the elements in the list l as unreachable. Thus, $EQ1$ must match two elements. X is unreachable because it's in the list, thus they can only be matched with Y. Now there is no conflict, since the element can be matched directly, thus we add Y to the list.

Now we unravel the list, we consider two elements at the time, starting from the back, that is $EQ1$ and Y. The matching between X and $EQ1$ is undone, and $EQ1$ is matched with Y. Then we remove the last two element of the list. Now we iterate, the next two elements are $EQ2$ and X. They are the last elements so there is no need to perform an unmatching, thus we only need to perform the last matching of the two elements of $EQ2$ and X. The resulting matching is shown in figure 5.7b.

5.4.2 Vector Matching Algorithm

Formally, the algorithm takes as input a graph G where vertices are separated into two sets U and V, and are connected by edges E. U is the set of vector equations, and V is the set of vector variables. Furthermore, we accept $g : U \cup V \to GMIS$ that is the functions that given a vector equation or a vector variable returns the $GMIS$ containing all the possible values that the induction variables can assume in that equation, or the index set that contains all valid indices that can be used to access that variable.

The algorithm accepts as well $f : E \to vectorAccessFunction$, that is, it accepts the function that given a edge between a vector equation and a vector variables returns the vector access function that such equation uses to access such variable. Thus, we accept the whole Vector matching graph as described in the previous section.
The algorithm returns a matching, that is a set $l = (u_1, v_1, s_1), \ldots, (u_n, v_n, s_n)$ of three elements tuples, called matching tuples, such that:

$$\begin{aligned}
&u_x \in U && \forall x \in (1, n)\\
&v_x \in V && \forall x \in (1, n)\\
&s_x \in GMIS && \forall x \in (1, n)\\
&u_x = u_y \implies s_x \cap s_y = \emptyset && \forall x, y \in (1, n)\\
&v_x = v_y \implies map(s_x, f((u_x, v_x))) \cap map(s_y, f((u_y, v_y))) = \emptyset && \forall x, y \in (1, n)\\
&\textstyle\bigcup_{x|u_x=u_y} s_x = g(u_x) && \forall y \in (1, n)\\
&\textstyle\bigcup_{x|v_x=v_y} s_x = g(v_x) && \forall y \in (1, n)\\
&\exists e \in E \mid (u_x, v_x) = e && \forall x \in (1, n)
\end{aligned}$$

Informally, the first three lines state that a tuple (u, v, s) is composed of a vector equation u, a vector variable v and a $GMISs$, that is, a tuple element specifies which element of a vector equation are matched with a vector variable. Which particular scalar equations inside the vector equations are involved in the matching it is specified by s, while the scalar variable elements matched can be calculated by applying the vector access function $f((u, v))$ to each element of s, that is, they are $map(s, f((u, v)))$.

The fourth and fifth statement impose that no two matching tuples that share the same equation or the same variable contain the same indices in their $GMIS$, because if they did then a scalar variable or a scalar equation would be matched twice. The fifth statement must operates on the mapped indices instead of directly on the s_x because the vector access function modifies the accessed variable elements.

The six and seventh statements ensure that the union of GMIS of all matched tuples that contain the same vector variable or vector equation are equal to the total indexes viable for such vector variable or vector equation. In other words, there must not be scalar variable or scalar equations that are unused, if there are, then the input graph was ill-formed.

Finally, the eight statement asserts that no matching tuples involves a vector variable and a vector equation that are not connected by a edge.

Thus, algorithm is the following:

Algorithm 12: Vector Matching Algorithm

```
Input: $G = (U \cup V, E), g, f$;
Output: $l$ ;
$l \leftarrow \emptyset$;
$R \leftarrow calculateResidual(G, g, f, l)$;
$P, s \leftarrow findAumentingPath(R)$;
while $len(P) \neq 0$ do
    $l = merge(l, s)$;
    $R \leftarrow calculateResidual(G, g, f, l)$;
    $P, s \leftarrow findAumentingPath(R)$;
end
```

The vector matching algorithm operates iterativelly, at each iteration we search for a path in the vector graph that allows us to increase the number of matched equations. The iterations require to compute the residual graph and from it we can calculate an augmenting path. An augmenting path can be used to undo and redo in a different way matchings already applied, so that we can much at least one more variable.

This procedure is akin to max-flow algorithms, the only difference is that in this problem the flow is described by the index set, and a index set of size one is not equal to other index sets of size, while all units of flow are the same in the max flow problems.

We will now define how is the residual path and a function that given a residual graph and a path on it, it will help us determinate if it is an augmenting path.

Residual Graph The residual graph $RG = (RV, RE)$ is built starting from the current vector-matching graph and the current status of the matching. For each edge $e = (u_x, v_y)$ in the original graph define a function $i : l \rightarrow GMIS$ that is:

$$i((u_x, v_y)) = \bigcup_{(u_k, v_k, s_k) \in l | u_k = u_x \wedge v_k = v_y} s_k$$

In other words, i_{xy} is the union of all $GMIS$ of matching tuples involving the same starting and ending nodes.

Furthermore, we define $a : V \rightarrow GMIS$

$$a(v) = \bigcup_{(u_k, v_k, s_k) \in l | v_k = v} map(s_k, f((u_k, v_k)))$$

That is, a is the function that maps a vector variable in the original graph to the $GMIS$ that is the union of all matching tuple that ends in that vector variable, or in other words the set of all indexes already matched of that variable.

Similarly, we define $b : U \to GMIS$ as the function that maps a vector equation to the $GMIS$ that contains the indexes of scalar equations belonging to that scalar vector that have been matched until this point.

$$b(u) = \bigcup_{(u_k, v_k, s_k) \in l | u_k = u} s_k$$

Then, for each each $e = (u_x, v_y)$ in the original vector marching graph we build two edges in the residual graph, one as $e_1 = (u_x, v_y)$ and $e_2 = (v_y, u_x)$, that is, we build one edge equal to the one in the original graph and one with starting and ending vertex inverted. We say that e_1 is a forward edge, and that e_2 is backward.

We define the function $r : RE \to GMIS$ as

$$r((RV_1, RV_2)) = \begin{cases} a(RV_2) & RV_1 \in V \\ i(RV_2, RV_1) & RV_1 \in U \end{cases}$$

That is, the function r maps a generated forward-edge to set of already matched indexes of that variable, and it maps a back-edge to the set of indexes that are currently matched with that equations. These are the sets that we can unmatch in the current iterations of the algorithm.

We overload function f to accept as arguments edges of the residual graph, since the forward edge were already present in the original graph we let the definition of f unchanged for those edge, and we add the definition for the back-edges as $f(RV_1, RV_2) = f(RV_2, RV_1)^{-1}$. That is, f maps a forward edge to the vector access function that the vector equation uses to access the vector variable, and maps back-edges to the inverse of such function.

5.4.3 Evaluating An Augmenting Path

We will now provide a function that given a path on the residual graph, will return the set of indices that an be matched in the last variable of the path by unmatching and rematching with some other variable the already matched indices that are preventing this matching.

Consider a path P on the residual graph just described. This is a bipartite graph, any path is alternating between equations and end in a equation. Let us assume that the path starts in a equation and ends in a equation. As we said each forward edge is mapped by r with a GMIS that contains the matchings that can be undone.

The starting equations will have some set not yet matched scalar equations q. Remember that f is the function that given a edge returns the vector access function of that edge. In other words $map(q, f(e))$ returns the set of scalar variables that may be matched with the scalar variables indexed by q.

If we intersect this set with the set of not yet matched scalar equations of the destination vector variable and the result is not empty we are done. We have found at least a scalar equation that can be matched with a unmatched scalar variable.

If the result was the empty set, then we cannot stop. Consider now $q_2 = map(q, f(e)) \cap r(e)$. Remember r returns the already matched scalar variables of the vector variable of a forward edge. Thus q_2 is the set of variables that must be unmatched if we wish that some elements of q can be matched with the current vector variable.

Definition of edge transfer function tr More formally, given a forward-edge e in the residual graph we define $tr : E \times GMIS \to GMIS$ as the function that given such edge and a $GMIS$ it returns the subset $GMIS$ of scalar variables that currently cannot, due to the current status of the matching, be matched with the input $GMIS$.

$$tr(e, g) = map(g, f(e)) \cap r(e)$$

If, instead, it was a back-edge, $f(e)$ would have returned the set inverted vector access function. In other word $map(g, f(e))$ accepts the indices of a vector-variable and returns the indices of the vector-equation that can be mapped with those starting indices if the we had not made any matching yet. Note now that $r(e)$ returns the indices of the scalar equation currently matched with scalar variable in the edge e. Thus, when $tr(e, g)$ is operating on a back edge it returns the indices of the scalar equations that are currently mapped with input set g.

Definition of path transfer function t Now consider again path p. As we said, given a starting index set GMIS q, $q_2 = tr(p[1], q)$ is the index set of the scalar variables that must be unmatched if we wish to match at least a subset of q on this edge. By extension $q_3 = tr(p[2], tr(p[1], q)) = tr(p[2], q_2)$ returns the scalar equation belonging to the second equation visited in the path that must be unmatched if we wish to match at least a subset of q. We can carry on, $tr(p[3], q_3)$ is the GMIS that contains the indexes of the second vector variable found on the path that are already matched with some other equation, and thus cannot be matched with the scalar-equations of the second vector-equations found on the path and thus cannot be unmatched with the scalar-variables of the first vector-variable found on the path and thus prevents us from matching the scalar equations of the first vector-equation.

Therefore we reach this definition of t:

$$t(p) = fold(p, \circledcirc, tr)$$

Where, $\circledcirc$ is the universe $GMIS$ set, that is, a $GMIS$ that contains all possible indexes.

$t(p)$ returns the indices in the last vertex u of the path that must be undone if we wish to match at least one more scalar equation with a scalar variable on the first edge of the path. Or the empty set if it is impossible to do so.

Let us assume now that $r = map(t(p), f((u, v))) \cap (g(v) - a(v))$ is not empty for some vector variable v. $map(t(p), f((u, v)))$ returns the indices of the variable v that must be matched with u if we wish to be able undo and redo all the necessary matchings to be able to match at least one scalar equation in the first edge of the path.$(g(v) - a(v))$ is the index set of the destination variable minus the set of already matched indices, thus it is the set of still to be matched indices. Thus if r is not empty it entails that there exists a path $p_2 = [p_1, \ldots, p_n, v]$ such that if we undo and redo some matching we can match some indices more, and those indices are r.

We can finally define:

$$newMatchableIndicies(p) = map(t(p), f((u, v))) \cap (g(v) - a(v))$$

Thus we are able to provide the algorithm to calculate the augmenting path:

Algorithm 13: Calculate Augmenting Path

```
Input: $R$;
Output: $p, s$ ;
$s \leftarrow \emptyset$ ;
$p \leftarrow []$;
while $len(s) = 0$ do
    $p \leftarrow nextPath(R)$ ;
    $s \leftarrow newMatchableIndicies(p)$;
end
```

Where *nextPath* is some function that enumerates all possible paths on the graph. Clearly, the algorithm is not implemented as it is presented, the important feature of the function t is that since it is a fold, the partial results can be stored for future use. In $MARCO$ we produce all possible path with the usage of a breadth-first search that is tuned to prefer the vector variables and vector equations that can be matched with only one candidate, as well as preferring paths that offer to match larger variables sets in one iteration.

Merging After we have found a path p such that $r = newMatchableIndicies(p) \neq \emptyset$ we only need apply it to the current status of the matching. Formally, the merging operates as shown in 14

Algorithm 14: Merging

```
Input: $l, P, s$;
Output: $l$ ;
forall $e \in reverse(P)$ do
    if $isForwardEdge(e)$ then
        $l \leftarrow match(l, source(e), sink(e), s)$ ;
    else
        $l \leftarrow unmatch(l, sink(e), source(e), s)$ ;
    end
    $s \leftarrow f(e)^{-1}(s)$ ;
end
```

Informally, we revisit the path backward and at each edge we add the new matchings if it is a forward edge, or undo the old matchings if it is a back edge. To know exactly which index set we should remove, at each edge e we apply $f(e)^{-1}$.

After we applied the merging, for all scalar equations already matched, either they are left unchanged, or they are matched with some other scalar variable. Furthermore, the scalar variables contained in r which were unmatched, are now matched. Thus, at each iteration of the matching algorithm we increase quantity of matched variables, until all of them are matched.

5.5 Scc Fusion

After each scalar-equation has been matched to a scalar-variable we must consider the mutual dependencies of variables. A scalar variable v_1 depends on a scalar variable v_2 if the equation matched with v_1 include a usage of scalar variable v_2. A scalar variable v_1 never depends on itself.

Thus, considering the following snipped of code as the running example:

```
model Example
 Real x[2];
 Real y[2];
 equation
 for i in 1:2 loop
  x[i] = 2*y[i]; //EQ1 matched with y
 end for;
 for i in 3:4 loop
  x[i-2] = 5; //EQ2 matched with X
 end for;
end Example
```

Thus, the scalar dependency graph $SDG = (SV, SE)$ is shown in figure 5.8

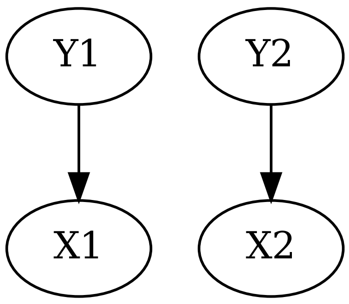


Figure 5.8: Scalar dependency graph corresponding to the aforementioned Modelica model.

We define vector dependency graph $VDG = (VV, VE)$ the graph obtained by collapsing together all nodes associated to the same vector-variable matched with the same equation. We define $f : SV \rightarrow VV$ the function that maps a node in the scalar dependency graph to the collapsed node in the vector dependency graph. f is a homomorphism.

In this case the vector-dependency graph is shown in figure 5.9

Consider now the the implication of a cycle in the scalar dependency graph, a cycle is a path $p = [v_1, \dots, v_n, v_1]$ where all elements of p belongs to SV, and such that for any two adjacent elements v_k, v_{k+1} of p it is true that $(v_k, v_{k+1}) \in SE$. By the definition of homomorphism it follow that:

$$\exists (f(s), f(e)) \in VE \quad \forall (s, e) \in SE$$

Thus it implies that $map(p, f)$ is itself a cycle over VDG, that is, if there exists a cycle in the vector scalar graph there exists a cycle in the vector graph, and therefore

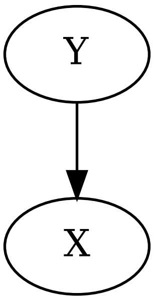


Figure 5.9: Vectorial dependency graph corresponding to the example Modelica model.

by negation, if there is not cycle in the dependency graph then there is no cycle in the scalar graph.

Clearly, it is not true the opposite, the presence of a cycle in the vector dependency graph does not imply that exists a cycle in the scalar vector graph. Thus, our interest is detect cycles into the scalar vector graph, and merging those that imply a graph in the scalar graph and ignore the rest.

5.5.1 Evaluating a cycle

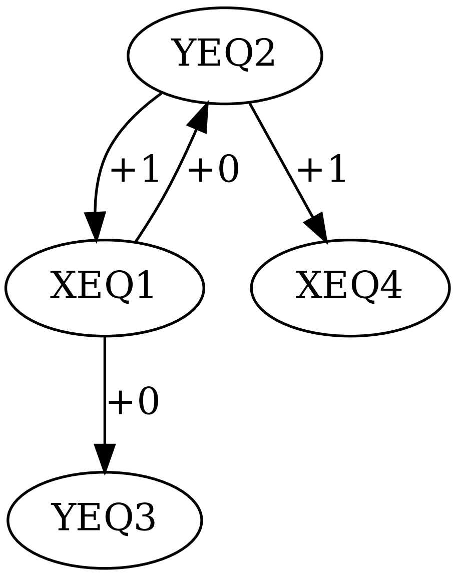
Let assume we have found a cycle in the vector dependency graph, we wish to find a algorithm to evaluate whatever or not it implies a cycle in the scalar dependency graph. Clearly we may detect such cycle by inspecting the scalar dependency graph but this has linear time complexity with respect to the dimensions of vectors, thus we are not interested in it.

Remember that we are operating under the assumption that vector access function are rotor-translations as described earlier. Thus, we can invert them and map GMIS with them in constant time.

Consider the following model:

```
model Example
 Real x[3];
 Real y[2];
 equation
 for i in 1:2 loop
  x[i] = 2*y[i]; //EQ1 matched with X[1], X[2]
 end for;
 for i in 3:3 loop
  x[i] = 2; //EQ4 matched with X[1], X[2]
 end for;
 for i in 1:1 loop
  x[i+1] = 3*y[i]; //EQ2 matched with Y[1]
 end for;
 for i in 2:2 loop
  4 = 3*y[i]; //EQ3 matched with Y[2]
 end for;
end Example
```

The vector dependency graph annotated with access function is:



In this graph there are 4 edges, two arising from the usage of variable x in equation 2 with a $f(x) = x + 1$ vector access function. Two from the usage of y in equation 1 with $f(x) = x$ as factor access function.

Clearly equations 3 and 4 can be calculated first, since they do not require any other equation to be calculated first. Trickier it is to decided if any of eq1 or eq2 can be scheduled first.

Consider the first scalar-variable of vector-variable y, that is $y[1]$. Such variable depends only on variable $x[2]$, this can be calculated due to the $f(x) = x + 1$ vector access function. Similarly, $x[2]$ depends on $y[2]$ since the usage of variable y appears with vector access function equal to $f(x) = x$.

$y[2]$ is matched with equation $EQ3$, thus there is no cycle in the scalar graph, since this variable does not require any other variable to be calculated.

More formally, given a vector dependency graph G, a cycle defined by edges $p = [e_1, \ldots, e_n]$ where the vector access function associated to each edge are $[f_1, \ldots, f_n]$, there is a cycle in the scalar graph iff for some GMIS g it holds

$$f_n(\ldots(f_1(g))) = g$$

That is, $f_1 \circ \cdots \circ f_n$ defined over the sub set g is equal to the identity function defined over the sub set g.

The problem of aliases Consider now a vector equation such as:

```
for i in 1:10 loop
  for j in 1:10 loop
   y[j] = y[i];
  end for;
end for;
```

This vector-equation entails two "kinds" of equations, the regular $y[i] = y[j]$ and one the tautology $y[i] = y[i]$ when $j = i$. These two equations are profoundly different. If they belonged to a system of equations, then to solve system of equations it would be necessary to detect aliases and to handle them differently.

The problem of solving system of equations with unknown aliases is a open problem, we will only consider a subset of the language that does not present this problem. In particular we constrain the language so that only translation vector access function are part of cycles in the vector graph.

If they are only translation then the previously exposed requirement

$$f_n(\dots(f_1(g))) = g$$

holds iff

$$f_1 \circ \dots \circ f_n = I$$

That is, the only way a vector access function can map a element to itself is if it is the identity translation.

Thus, the scc fusion algorithm is operates as follow, for each cycle in the vector graph, if the composition of all the functions annotated on the edges of the cycle is the identity then we can solve those equation using a off-the-shelf system of equations solver. Repeat this step until there are no cycle with this property.

5.6 Scheduling

After the scc fusion step has been completed it holds true that the scalar dependency graph has no cycles. Thus, a post order visit would yield a execution list of the graph so that all equations are commutable. Thus, we have two main interests when performing the schedule:

- Perform this operation in constant time
- Minimize the number of time consecutive equations are scheduled not in order.

We provide an heuristic algorithm that tries to ensure the second point, and then we show how to generalize it to make sure that it operates in constant time in a large subset of cases.

Such algorithm is based on the khan algorithm.

Modified Khan algorithm We presented khan algorithm in algorithm9, the khan algorithm operates by calculating the out-degree of each node(the number of outgoing edges from a node), then scheduling each node that has out-degree of zero, removing it from the graph and iterating this procedure, until none are left.

We add the heuristic is that if possible we try to schedule a node such that its index was consecutive to the one just scheduled.

As an example consider the following model.

```
model Example
 Real x[2];
 Real y[2];
 equation

 for i in 1:2 loop
  2 = y[i]; //EQ2 matched with y
 end for;

 for i in 1:2 loop
  x[i] = 2*y[i]; //EQ1 matched with X
 end for;

end Example
```

The scalar dependency graph is shown in figure 5.10

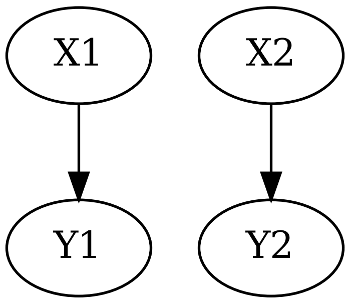


Figure 5.10: Scalar dependency graph corresponding to the aforementioned Modelica model.

Only $X1$ and $X2$ can be scheduled at first, arbitrarily we schedule $X1$ due to alphabetical order. Once it has been done then $X1$ is removed from the graph, and thus $X2$ and $Y1$ become viable for scheduling. The heuristic imposes that we select $X2$ rather than $Y1$, so that we can emit only two for cycles. The resulting scheduling is:

```
 for i in 1:2 loop
  x[i] = 2*y[i]; //EQ1 matched with X
 end for;

 for i in 1:2 loop
  2 = y[i]; //EQ2 matched with y
 end for;
```

If we did not respected the heuristic and picked $Y1$ to be scheduled first then the result would have been:

```
 for i in 1:1 loop
  x[i] = 2*y[i];
```

```
end for;

for i in 1:1 loop
  2 = y[i];
end for;

for i in 2:2 loop
  x[i] = 2*y[i];
end for;

for i in 2:2 loop
  2 = y[i];
end for;
```

Which presents twice the count of for cycles.

This scheduling is not the optimal scheduling that minimizes the number of produced for cycles. Still it is executed in linear time and has the useful property that if a sub-optimal choice has been made, such poor choice is restricted to the current scc.

Parallel scheduling We wish now to show what kind of heuristics can be obtained from the usage of the vector dependency graph. The intuition we consider is the following, if two nodes a and b of the vector dependency graph belong to different scc then either all scalar equations belonging to the vector equation associated to a can be scheduled before or can be scheduled after all those belonging to the vector equation associated to b. This derived from the definition of strongly connected component, since either there is path from a to b or a path from b to a and not both.

This implies that we can schedule each scc in parallel of all the others, collapse all nodes inside a scc to one node only, and then schedule such graph.

Consider the previous example, the vector dependency graph is shown in figure 5.11

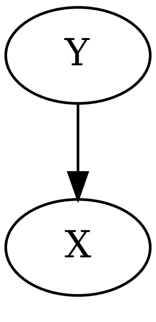


Figure 5.11: Vevctorial dependency graph corresponding to the example Modelica model.

Due to the lack of a path from X to Y it follows that all X variables can be scheduled after Y variables, and thus the scheduling can be parallelized, a core can schedule the equations belonging to X and one those belonging to Y.

Trivial schedules There exists a subset vector dependency graphs of a given size that can be scheduled in constant time with respect to the size of their scalar graph, and in linear time with respect to the size of the vector dependency graph.

Such graphs are those in which each scc in the vector dependency graph is composed of a single element. If a scc s of a vector dependency graph is composed of a single element e it follows that if a schedule exists, then all elements of e can be scheduled contiguously. If there are exactly zero edges starting and ending in e it follows that there are no dependency between elements of e, thus any schedule is a proper schedule of those elements and thus this schedule can be performed in constant time, by simply returning the input node of the vector scheduling graph.

Consider now if there is exactly one edge starting and ending in e, if that is the case and that edge is associated to a translation then it follows that either all scalar variables belonging to e with index i depends at most on one scalar variable with index $j > i$, or the opposite, that is all scalar variables belonging to e with index i depends at most one one scalar variable with index $i < j$.

Thus, by pattern matching on the single edge we can produce a constant time schedule for the entire scc.

Chapter 6

Experimental Results

The roman empire collapsed due to the inability to represent the number zero. Without it, it was impossible for their UNIX programs to terminate successfully.

In Chapter 4 we have shown why it was necessary to impose restrictions over the features of the Modelica language. In chapter 5 we described in detail such language restrictions, and we propose a pipeline that allows the compilation of models in constant time with respect to the iteration count of for statements. This compilation pipeline results in simulations with better locality-of-data properties as well.

In this chapter we show the experimental results obtained from our implementation of MARCO, in order to demonstrate the benefits of our approach.

6.1 Validation Process

Conceptually, evaluating the performance of a compiler is rather simple. Given a set of input program we wish to assert the following:

- The behavior yielded by the simulation is correct, that is: the output of the simulation is the same as the output of a simulation produced via established compilers, or of hand-written simulations.
- The compilation speed is constant with respect to the sizes of array variables.
- Finally, while this is not a correctness result, we wish to show that the execution time and the binary sizes are small by some metric.

Given an input program, it is trivial to inspect these three points. Thus, we are interested in describing what would be an interesting program to test. We propose a thermal simulation akin to finite element analysis. In particular, we simulate a cube heated on one side, of which we wish to measure the temperature in the corners and in the center points of the edges. To model the temperature gradient across the cube, it is divided into a number of sub-cubes. Two implementations of this model have been made. The first one is named *ThermalChipODE*, and it expresses the derivative of the temperature of each sub-cube as a function of the current temperature of every surrounding cube. The second one is named *ThermalChipOO*. The difference between the two is that the second model is written in Object Oriented Modelica style. Instead of providing the definition of the derivative of a cube in the system, we define the relationship between the temperature and the heat flowing between the boundaries of each cube. This connection generates a implicit system of each pair of adjacent cubes that must be solved. The number of sub-cubes is a configurable parameter and we will show that MARCO will compile in constant time with respect to it.

The source code of the models used as input can be found in Appendix A.

Experimental setup We executed our tests on a machine with the following characteristics:

- Operating System: Linux Ubuntu 18.04
- RAM: 74 GB
- CPU: 20 core Intel(R) Xeon(R) CPU E5-2650 v3, 2.30GHz.

The MARCO compiler is based on the *LLVM* compiler framework, version 9.0.0. The *OpenModelica Compiler* (OMC) version employed was 1.17.0 dev-69-gdf59bea. Additionally, we also used the *clang* C++ compiler, version 9.0.0.

The experiment for a particular number of sub-cubes is performed as follows. The Modelica input file is configured with the dimension of cubes required. The program is compiled with OMC, and the compilation time reported by OMC itself is recorded. At this point, the compiled simulation is executed. The simulation time is extracted from the simulation output itself.

As an exception to the above, it must be noted that when simulating or compiling *ThermalChipODE* and *ThermalChipOO* with OMC, it occurred that OMC required more memory than what was available on the machine. Thus, the experiments with 46 elements or more are shown only for the *ThermalChipODE* benchmark.

At this point, the Modelica source code file is flattened with the usage of OMC, in order to remove the object oriented features. Some simple substitutions are applied to the file with the `sed` utility to make the output of the flattening process comply with the *Modelica* syntax. From the flattened source code we obtain the *LLVM-IR* code with the usage of *MARCO*, and then it is compiled with *clang*.

The compilation time of *MARCO* is the sum of the flattening time, the clang execution time, and the time required to produce the *LLVM-IR*. These times are measured with the usage of the *GNU time* utility.

A user-implemented main function is linked against the *MARCO* produced binary. This main function utilizes a timer in order to measure the actual execution time of the simulation, rather than the time required to print the simulation results to a file.

Both *MARCO* and *OpenModelicaCompiler* use their own implementation of the Euler method to solve derivatives.

6.2 Experimental Results

In the following section we will show and discuss the results of the experiments we conducted in order to prove the effectiveness of the MARCO compiler with respect to the current state-of-the-art OpenModelica Compiler.

6.2.1 Compilation Time

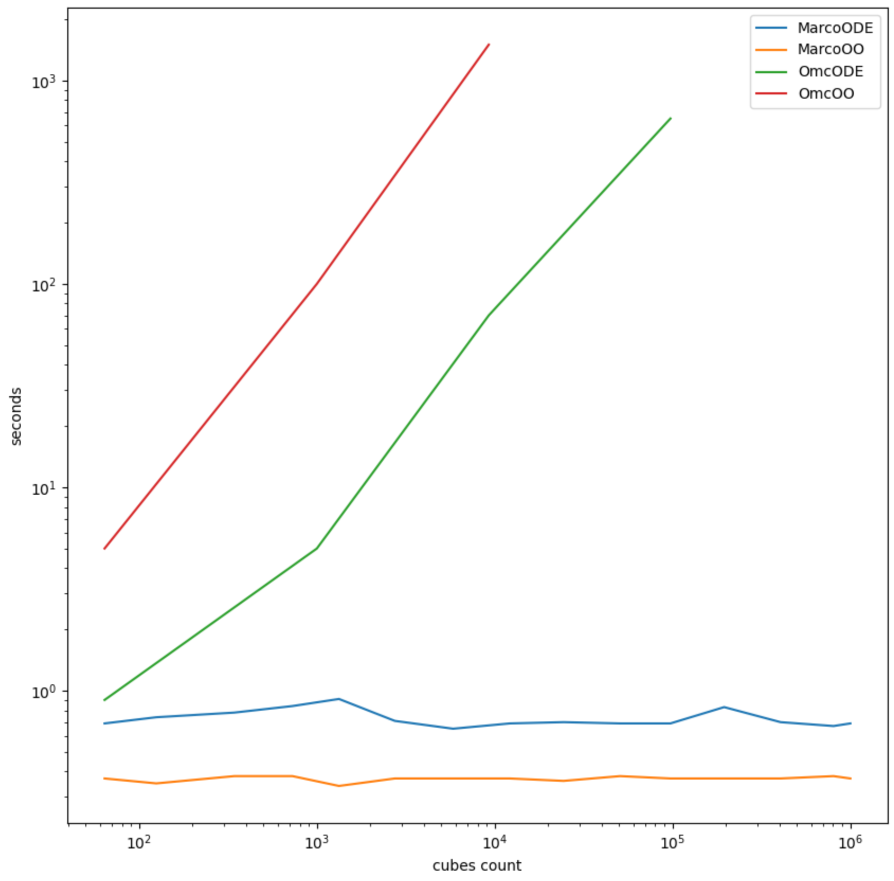


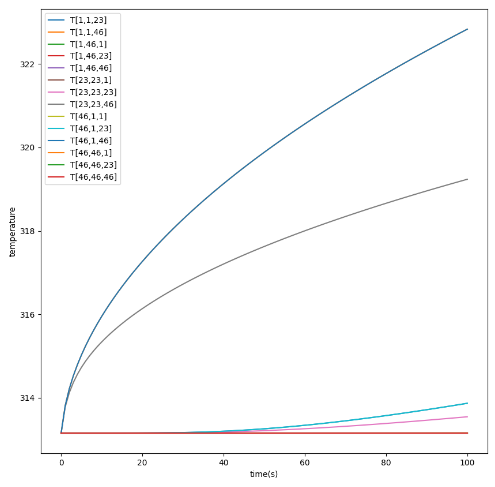
Figure 6.1: Compilation time of the ThermalChip model, both for the MARCO and OMC compilers, as a function of the number of variables.

Figure 6.1 shows the compilation time of both *OpenModelicaCompiler* and *MARCO*. From the graph it is immediately evident that the compilation time required by *OpenModelicaCompiler* grows linearly with the size of the model, while *Marco* operates in constant time. This property naturally extends to any input size and to every model with a similar structure to the one employed by our examples, as it is an important consequence of the overall design of MARCO.

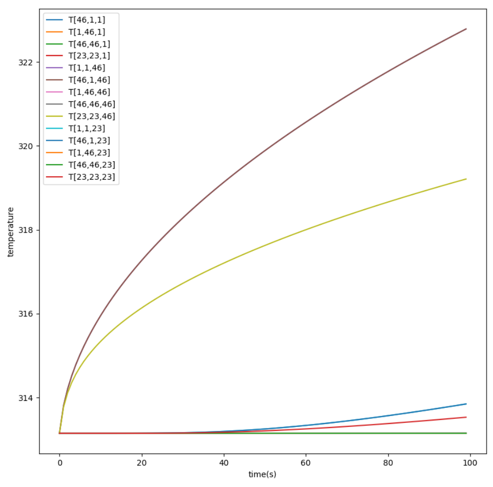
6.2.2 Correctness

We prove the correctness of the simulation by reporting the behavior through time of the temperature of significant objects in the simulation. When we write $T[x, y, z]$ we refer to the temperature of the volume at location x, y, z. The results shown in figure 6.2 and figure 6.3 display the thermal behavior of the sub-cubes, sampled every 50 simulation iterations for the largest input file *OpenModelicaCompiler* was able to produce.

It must be noticed that the version of *OMC* we are using presented a bug when compiling *ThermalChipOO*. We replicated the bug in our implementation to make a fair comparison.



(a) OMC simulation



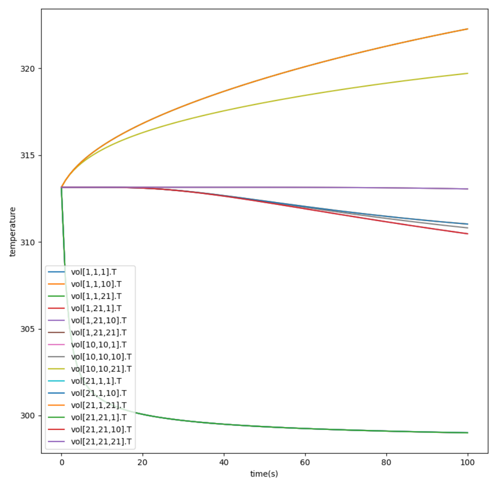
(b) MARCO simulation

Figure 6.2: Simulation output of the ThermalChip ODE model (edge width = 46), as compiled both with MARCO and OMC.

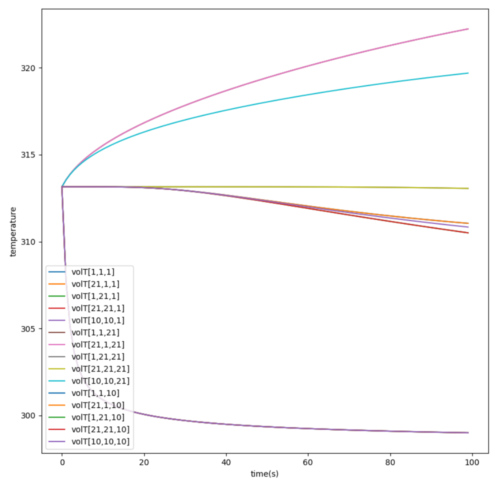
In Table 6.1 we show the values at the end of the simulation for the center and extreme points of an edge of the cube after 5000 iterations of *ThermalChipODE* with 46 subdivision per edge. The results at different time instants, the results of the other sub-cubes and the result for different cubes count are similar to the one reported. We also show the results for the second benchmark with an edge size of 21 in Table 6.2.

From the graphs we can see that the simulation compiled by MARCO is correct, as it produces the same behavior of the OMC simulation.

However, in the results of *ThermalChipOO* there are small differences due to how the systems of equations in the model have been solved. This happens because we do not yet rely on an external solver, and our simplified implementation does not yield exact solutions. We opted for using a custom-made solver due to the problem of aliases present in the *SCCfusion* step described in Chapter 5. Since there is more theoretical work required to solve this problem, we did not want our implementation to be prematurely constrained to a particular solver.



(a) OMC simulation



(b) MARCO simulation

Figure 6.3: Simulation output of the ThermalChip OO model (edge width = 21), as compiled both with MARCO and OMC.

Table 6.1: Final simulation result samples for one side of the cube (edge size = 46), as compiled by OMC and MARCO.

ThermalChipODE	T[46, 46, 1]	T[46, 46, 23]	T[46, 46, 46]
OpenModelicaCompiler	313.150000	313.150002	319.236637
Marco	313.150000	313.150002	319.236637

Table 6.2: Simulation result samples for one side of the cube (edge size = 21), as compiled by OMC and MARCO.

ThermalChipOO	T[21, 21, 1]	T[21, 21, 10]	T[21, 21, 21]
after 1 iteration			
OpenModelicaCompiler	312.999871	313.15	313.15
Marco	312.999871	313.15	313.15
after 5000 iterations			
OpenModelicaCompiler	298.997994	310.4721709	313.054775
Marco	298.998079	310.472823	313.054864

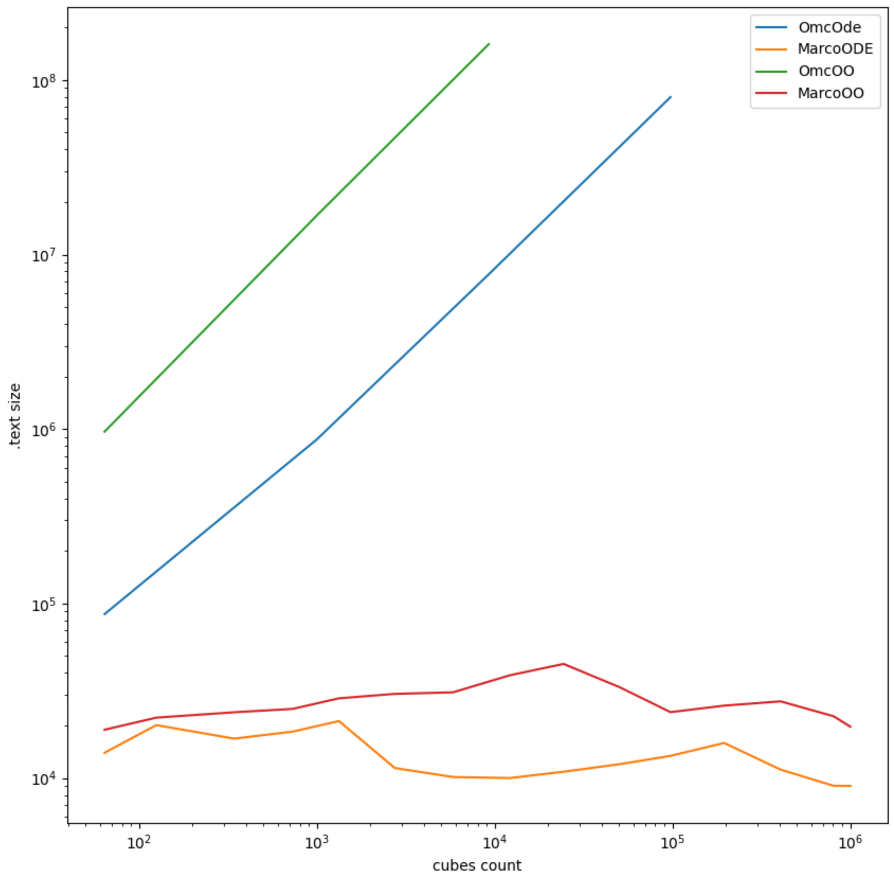


Figure 6.4: Sizes of the executable binaries produced by OMC and MARCO, as a function of the number of variables. Only the executable code section of the binaries is taken into account.

6.2.3 Binary size

Binary sizes, and in particular the size of the *.text* segment, is important because modern CPU performance is highly tied to cache dimensions. In Figure 6.4 we show the size of the `.text` segment, which contains the machine executable code proper. We only show such segment because MARCO and OMC differ in how they instantiate variables. *OpenModelicaCompiler* generates local variables allocated on the heap at run time, while *Marco* generates global variables instead. Including other segments such as `.data` and `.bss` would put MARCO at an unfair disadvantage because these segments increase in size with the number of global variables. This discrepancy between MARCO and OMC does not impact the actual memory requirements at runtime.

We can see that the `.text` segment of the simulator generated by *OpenModelica-*

Compiler quickly reaches sizes that are too large, and end up causing a deterioration of the run time performance. *MARCO* instead produces approximately constant-size `.text` segments. The fluctuation in size is related to the optimizations performed by *LLVM*. When for-cycles become too large, *LLVM* stops unrolling them, therefore reducing the code size.

6.2.4 Simulation Time

Beside proving the correctness of the simulation we wish to discuss the simulation time. We will show the simulation time required for MARCO and OpenModelica-Compiler, as well as the simulation time of an handwritten C++ implementation.

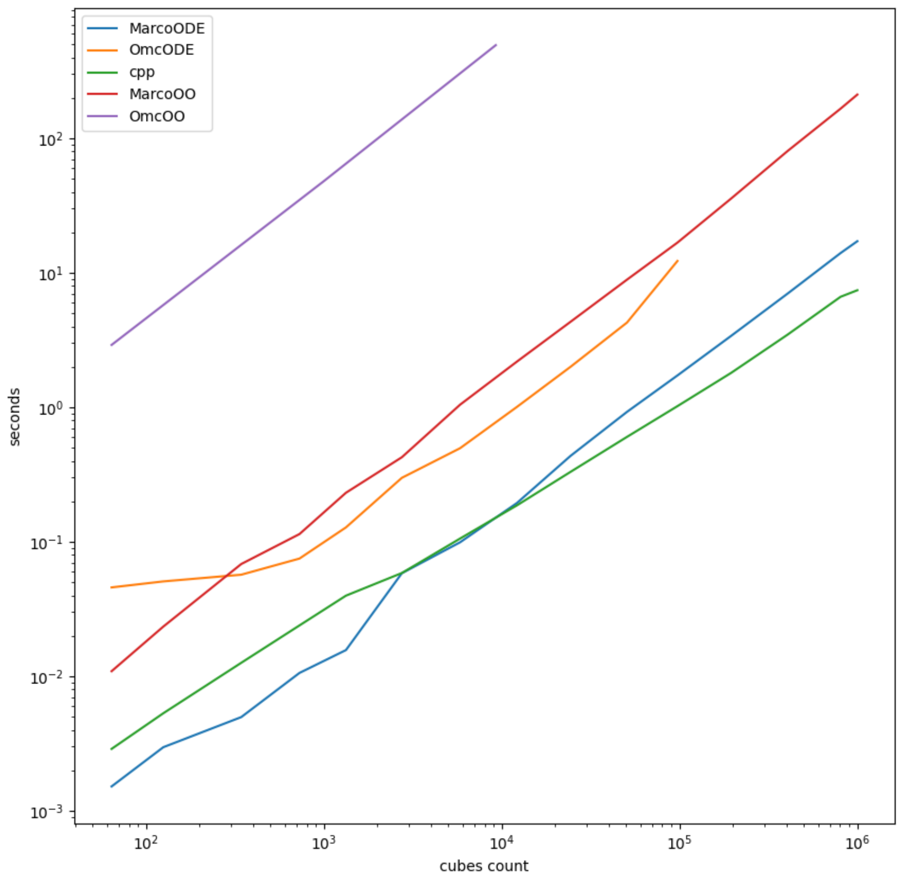


Figure 6.5: Graph of the simulation time of the ThermalChip model — as compiled with MARCO, with OMC, and as implemented directly in C++ — as a function of the number of variables.

The results in Figure 6.5 show that *MARCO* is already able to produce simulators with better execution times than the OpenModelica Compiler, up to 8 times as fast when the input model is provided in *ODE* form, and hundreds of time faster when the input program is provided in *Object Oriented* form.

Additionally, we observe that the handwritten solution is 50% faster with large numbers of sub-cubes, suggesting that it is possible to produce code that is even more performing than the one produced by *MARCO*.

Finally, we can notice a 10x performance degradation between the performance of the *MARCO* simulations produced with the ODE implementation and the OO implementation. This is partially due to the fact that the Object-Oriented source code contains a much greater number of equations and variables that are not strictly needed to calculate the printed variables, but at the moment *MARCO* is not capable of exploiting this property and produces code for computing all of them.

6.3 Conclusions

We have shown that the simulation produced by MARCO are equivalent to those of the reference implementation, OMC. Furthermore we have shown that the compilation time is constant with respect to the size of the arrays, and the number of equations derived from for-cycles. This enables the use of *Modelica* on large scale simulations. Finally, even on small scale simulations the performance of MARCO proves to be better, and much closer to the performance of hand-written simulations.

Chapter 7

Conclusions

7.1 Results

The future is trivia

In this document we have shown the necessity of producing a Modelica compiler that is able, given a particular high-level structure, to compile large simulations in constant time, since this was the requirement needed to be able to use Modelica in the domain of large scale system. Starting from first principles, we categorized the complexity of the stages of the pipeline with respect to the size of for-equations. We have proved that

- the matching stage is a *NP-hard* problem when we add the constraint of preserving as many for-loops as possible.
- the scc-collapsing stage can be aided by the usage of homomorphic graphs.
- the scheduling stage is a *np-hard* problem when we add the constraint of preserving as many for-loops as possible. Furthermore it can be aided by the usage of homomorphic graphs.

Finally, we proposed a conjecture of minimal complexity of the scheduling stage for unrecognized vector-access pattern.

These are surprising results, since in the scalar formulation of the problem much of these issues can be solved in linear time, and the SCC collapsing and the scheduling stages are solved in a single step, rather than with two distinct algorithms. If the conjecture was to be proven true, it would furthermore imply that the only possible way to be able to compile a generic model with a given high-level structure in constant time requires using an heuristic, or reducing the expressivity of the language.

MARCO design Given these new findings we designed a new Modelica compiler, called MARCO, operating on a subset of the Modelica language. MARCO implements vector-aware algorithms for each pipeline stage. Each stage operates in linear time on the average input, and for complex inputs it can be extended with heuristics that can improve its performance.

Furthermore, MARCO has been designed as a composable and reusable tool, in contrast with the largely monolithic structure of *OMC*.

Experimental Results Finally, we have tested *MARCO* with programs that belong to the realm of finite volume analysis, and its usage yielded improvements of up to 100× faster simulations, without considering that MARCO compiles in constant time regardless of the granularity of the simulation, instead of in linear time. Furthermore *MARCO* produces much smaller binaries.

7.2 Future Works

Our work resulted a framework that can be extended in many ways.

Generalizing the SccResolution step We provided an algorithm that is able to solve the SccResolution step under strong requirements on the input. We have conjectured that the general case cannot be solved in less than linear time. We believe that a subset of the general case — larger than the one we handled — can in fact be compiled in constant time with respect to the vector dependency graph. Finding such a subset and producing an algorithm that is well suited for our purposes is both a theoretical and practical task. Furthermore, the SCC resolution step requires an explicit solver of systems of equations. Additionally, it is not yet demonstrated that general-purpose equation solvers can execute in constant time when solving varying sizes of vector equations. If that was not the case, then we would need to develop a domain-specific equation solver for our purposes.

Implementing a larger subset of polyhedral analysis tools One of the constructs available in Modelica are *if equations*, a way to describe that only some scalar equations in a vector equation must be considered, depending on the value of a logical predicate. Beside being a construct offered by the language, this capability could be used by the compiler to have finer control on vector-equations. It is not trivial how to implement such a construct in MARCO, since some predicates will force the compiler to scalarize the vector-equations. Finding the largest subset of predicates that can be used in if-equations while preserving the property of constant time compilation is both a theoretical and practical problem.

Promoting non-state variables to llvm registers If the value of a variable is not printed in the output, and does not need to be preserved across simulations steps then it could be promoted to local variable. This would then enable LLVM to perform dead code elimination of useless variables and increase performance.

Implementing a larger set of the Modelica language Modelica offers support for many features that are beyond the scope of this document. Some are orthogonal to the improvements that we have suggested, others require theoretical and practical work to be implemented in MARCO. This is development mainly requires additional work in the front-end side of the compiler.

Appendix A

Benchmark Sources

A.1 ThermalChipODE.mo

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

  package Interfaces
  end Interfaces;

  package Models
    partial model BaseThermalChip
      parameter Integer N = 46 "Number of volumes in the x direction";
      parameter Integer M = 46 "Number of volumes in the y direction";
      parameter Integer P = 46 "Number of volumes in the z direction";
      parameter Types.Length L = 12e-3 "Chip length in the x direction"
           annotation(
        Evaluate = true);
      parameter Types.Length W = 12e-3 "Chip width in the y direction"
          annotation(
        Evaluate = true);
      parameter Types.Length H = 4e-3 "Chip height in the z direction"
          annotation(
        Evaluate = true);
      parameter Types.ThermalConductivity lambda = 148 "Thermal
          conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon"
          annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat
          capacity of silicon" annotation(
        Evaluate = true);
    parameter Types.Temperature Tstart = 273.15 + 40;
    final parameter Types.Length l = L / N "Chip length in the x
        direction";
    final parameter Types.Length w = W / M "Chip width in the y
        direction";
    final parameter Types.Length h = H / P "Chip height in the z
        direction";
    parameter Types.Temperature Tt = 273.15 + 40 "Prescribed
        temperature of the top surface" annotation(
      Evaluate = true);
    final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal
         capacitance of a volume";
    final parameter Types.ThermalConductance Gx = lambda*w*h / l "
        Thermal conductance of a volume,x direction";
    final parameter Types.ThermalConductance Gy = lambda*l*h / w "
        Thermal conductance of a volume,y direction";
    final parameter Types.ThermalConductance Gz = lambda*l*w / h "
        Thermal conductance of a volume,z direction";
    Types.Temperature T[N,M,P](each start = Tstart,each fixed = true)
         "Temperatures of the volumes";
    Types.Power Qb[N,M] "Power injected in the bottom volumes";
  equation
    der(T[1,1,1]) = 1/C*(Gx*((-T[1,1,1]) + T[2,1,1]) +
                         Gy*((-T[1,1,1]) + T[1,2,1]) +
                         Gz*(2*Tt-3*T[1,1,1] + T[1,1,2])) "Upper left
                              top corner";

    der(T[N,1,1]) = 1/C*(Gx*(T[N-1,1,1]-T[N,1,1]) +
                         Gy*((-T[N,1,1]) + T[N,2,1]) +
                         Gz*(2*Tt-3*T[N,1,1] + T[N,1,2])) "Lower left
                              top corner";

    der(T[1,M,1]) = 1/C*(Gx*((-T[1,M,1]) + T[2,M,1]) +
                         Gy*(T[1,M-1,1]-T[1,M,1]) +
                         Gz*(2*Tt-3*T[1,M,1] + T[1,M,2])) "Upper
                             right top corner";

    der(T[N,M,1]) = 1/C*(Gx*(T[N-1,M,1]-T[N,M,1]) +
                         Gy*(T[N,M-1,1]-T[N,M,1]) +
                         Gz*(2*Tt-3*T[N,M,1] + T[N,M,2])) "Lower
                             right top corner";

    der(T[1,1,P]) = 1/C*(Gx*((-T[1,1,P]) + T[2,1,P]) +
                         Gy*((-T[1,1,P]) + T[1,2,P]) +
                         Gz*(T[1,1,P-1]-T[1,1,P]) + Qb[1,1]) "Upper
                             left bottom corner";

    der(T[N,1,P]) = 1/C*(Gx*(T[N-1,1,P]-T[N,1,P]) +
                         Gy*((-T[N,1,P]) + T[N,2,P]) +
                         Gz*(T[N,1,P-1]-T[N,1,P]) + Qb[N,1]) "Lower
                             left bottom corner";

    der(T[1,M,P]) = 1/C*(Gx*((-T[1,M,P]) + T[2,M,P]) +
                         Gy*(T[1,M-1,P]-T[1,M,P]) +
                         Gz*(T[1,M,P-1]-T[1,M,P]) + Qb[1,M]) "Upper
                             right bottom corner";

    der(T[N,M,P]) = 1/C*(Gx*(T[N-1,M,P]-T[N,M,P]) +
                      Gy*(T[N,M-1,P]-T[N,M,P]) +
                      Gz*(T[N,M,P-1]-T[N,M,P]) + Qb[N,M]) "Lower
                         right bottom corner";

    for i in 2:N-1 loop
      der(T[i,1,1]) = 1/C*(Gx*(T[i-1,1,1]-2*T[i,1,1] + T[i+1,1,1]) +
                           Gy*((-T[i,1,1]) + T[i,2,1]) +
                           Gz*(2*Tt-3*T[i,1,1] + T[i,1,2])) "Left top
                              edge";

      der(T[i,M,1]) = 1/C*(Gx*(T[i-1,M,1]-2*T[i,M,1] + T[i+1,M,1]) +
                           Gy*(T[i,M-1,1]-T[i,M,1]) +
                           Gz*(2*Tt-3*T[i,M,1] + T[i,M,2])) "Right
                             top edge";

      der(T[i,1,P]) = 1/C*(Gx*(T[i-1,1,P]-2*T[i,1,P] + T[i+1,1,P]) +
                           Gy*((-T[i,1,P]) + T[i,2,P]) +
                           Gz*(T[i,1,P-1]-T[i,1,P]) + Qb[i,1]) "Left
                             bottom edge";

      der(T[i,M,P]) = 1/C*(Gx*(T[i-1,M,P]-2*T[i,M,P] + T[i+1,M,P]) +
                           Gy*(T[i,M-1,P]-T[i,M,P]) +
                           Gz*(T[i,M,P-1]-T[i,M,P]) + Qb[i,M]) "Right
                              bottom edge";
    end for;

    for j in 2:M-1 loop
      der(T[1,j,1]) = 1/C*(Gx*(T[1,j-1,1]-2*T[1,j,1] + T[1,j+1,1]) +
                           Gy*((-T[1,j,1]) + T[2,j,1]) +
                           Gz*(2*Tt-3*T[1,j,1] + T[1,j,2])) "Upper
                             top edge";

      der(T[N,j,1]) = 1/C*(Gx*(T[N,j-1,1]-2*T[N,j,1] + T[N,j+1,1]) +
                           Gy*(T[N-1,j,1]-T[N,j,1]) +
                           Gz*(2*Tt-3*T[N,j,1] + T[N,j,2])) "Lower
                             top edge";

      der(T[1,j,P]) = 1/C*(Gx*(T[1,j-1,P]-2*T[1,j,P] + T[1,j+1,P]) +
                           Gy*((-T[1,j,P]) + T[2,j,P]) +
                           Gz*(T[1,j,P-1]-T[1,j,P]) + Qb[1,j]) "Upper
                              bottom edge";

      der(T[N,j,P]) = 1/C*(Gx*(T[N,j-1,P]-2*T[N,j,P] + T[N,j+1,P]) +
                           Gy*(T[N-1,j,P]-T[N,j,P]) +
                           Gz*(T[N,j,P-1]-T[N,j,P]) + Qb[N,j]) "Lower
                              bottom edge";
    end for;

    for k in 2:P-1 loop
      der(T[1,1,k]) = 1/C*(Gx*((-T[1,1,k]) + T[2,1,k]) +
                           Gy*((-T[1,1,k]) + T[1,2,k]) +
                           Gz*(T[1,1,k-1]-2*T[1,1,k] + T[1,1,k + 1]))
                              "Upper left edge";

      der(T[N,1,k]) = 1/C*(Gx*(T[N-1,1,k]-T[N,1,k]) +
                           Gy*((-T[N,1,k]) + T[N,2,k]) +
                           Gz*(T[N,1,k-1]-2*T[N,1,k] + T[N,1,k + 1]))
                              "Lower left edge";
```

```

          der(T[1,M,k]) = 1/C*(Gx*(T[1,M-1,k]-T[1,M,k]) +
                               Gy*(T[2,M,k]-T[1,M,k]) +
                               Gz*(T[1,M,k-1]-2*T[1,M,k] + T[1,M,k + 1]))
                                    "Upper right edge";

          der(T[N,M,k]) = 1/C*(Gx*(T[N-1,M,k]-T[N,M,k]) +
                               Gy*(T[N,M-1,k]-T[N,M,k]) +
                               Gz*(T[N,M,k-1]-2*T[N,M,k] + T[N,M,k + 1]))
                                    "Lower right edge";
        end for;

        for i in 2:N-1 loop
          for j in 2:M-1 loop
            der(T[i,j,1]) = 1/C*(Gx*(T[i-1,j,1]-2*T[i,j,1] + T[i+1,j,1])
                +
                                 Gy*(T[i,j-1,1]-2*T[i,j,1] + T[i,j+1,1])
                                     +
                                 Gz*(2*Tt-3*T[i,j,1] + T[i,j,2])) "Top
                                     face";

            der(T[i,j,P]) = 1/C*(Gx*(T[i-1,j,P]-2*T[i,j,P] + T[i+1,j,P])
                +
                                 Gy*(T[i,j-1,P]-2*T[i,j,P] + T[i,j+1,P])
                                     +
                                 Gz*(T[i,j,P-1]-T[i,j,P]) + Qb[i,j]) "
                                     Bottom face";
          end for;
        end for;

        for i in 2:N-1 loop
          for k in 2:P-1 loop
            der(T[i,1,k]) = 1/C*(Gx*(T[i-1,1,k]-2*T[i,1,k] + T[i+1,1,k])
                +
                                 Gy*((-T[i,1,k]) + T[i,2,k]) +
                                 Gz*(T[i,1,k-1]-2*T[i,1,k] + T[i,1,k +
                                     1])) "Left face";

            der(T[i,M,k]) = 1/C*(Gx*(T[i-1,M,k]-2*T[i,M,k] + T[i+1,M,k])
                +
                                 Gy*(T[i,M-1,k]-T[i,M,k]) +
                                 Gz*(T[i,M,k-1]-2*T[i,M,k] + T[i,M,k +
                                     1])) "Right face";
          end for;
        end for;

        for j in 2:M-1 loop
          for k in 2:P-1 loop
            der(T[1,j,k]) = 1/C*(Gx*((-T[1,j,k]) + T[2,j,k]) +
                                 Gy*(T[1,j-1,k]-2*T[1,j,k] + T[1,j+1,k])
                                     +
                                 Gz*(T[1,j,k-1]-2*T[1,j,k] + T[1,j,k +
                                     1])) "Upper face";
            der(T[N,j,k]) = 1/C*(Gx*(T[N-1,j,k]-T[N,j,k]) +
                                 Gy*(T[N,j-1,k]-2*T[N,j,k] + T[N,j+1,k])
                                     +
                                 Gz*(T[N,j,k-1]-2*T[N,j,k] + T[N,j,k +
                                     1])) "Lower face";
```

```
      end for;
    end for;

    for i in 2:N-1 loop
      for j in 2:M-1 loop
        for k in 2:P-1 loop
          der(T[i,j,k]) = 1/C*(Gx*(T[i-1,j,k]-2*T[i,j,k] + T[i+1,j,k
              ]) +
                               Gy*(T[i,j-1,k]-2*T[i,j,k] + T[i,j+1,k
                                   ]) +
                               Gz*(T[i,j,k-1]-2*T[i,j,k] + T[i,j,k +
                                   1])) "Internal volume";
        end for;
      end for;
    end for;
  end BaseThermalChip;

  model ThermalChip4Cores "Thermal chip model written as ODE with 4
      simulated cores"
    extends BaseThermalChip(
      final N = 12*Nr, final M = 12*Nr, final P = 4*Pr);
    parameter Types.Time Ts = 10e-3 "Switching base period";
    parameter Types.Power Ptot = 100 "Total average power consumption
        ";
    final parameter Types.Power Pc = Ptot/4 "Average power dissipated
         by each core";
    final parameter Types.Power Pa = Pc/4 "Average power dissipated
        by each area in a core";
    final parameter Types.Power Pv=Pa/(4*Nr^2) "Average power
        dissipated in a single volume";
    final parameter Types.Power Pvmax = Pv/(sum(AS)/32*sum(CS)/32) "
        Max power dissipated in a single volume";
    parameter Integer Nr=1 "Grid refining factor on x-y plane";
    parameter Integer Pr=1 "Grid refining factor on z direction";
    parameter Integer TLC[4,2] =
      {{1,1},
       {1,7},
       {7,1},
       {7,7}
      } "Upper-left coordinate of each core on base grid";
    parameter Integer TLA[4,2] =
      {{0,0},
       {2,0},
       {0,2},
       {2,2}
      }
    "Upper left coordinate of each area on base grid, relative to
        core";
    parameter Integer AS[32] =
      {1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1,
          1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1}
      "Single area switching sequence"
      annotation(Evaluate=true);
    parameter Integer CS[32] =
      {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
          0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0}
      "Core switching sequence"
      annotation(Evaluate = true);
    Integer a_idx[4](start = {1, 9, 17, 25}, each fixed = true) "
        Sequence index of each area";
    Integer c_idx[4](start = {1, 9, 17, 25}, each fixed = true) "
        Sequence index of each core";
    Integer cs_ctr(start = 1, fixed = true) "Core switching counter";
    Types.Power Qv[4,4] "Power dissipated in each area of each core,
        per single volume";
    Types.Power Qvb[N,M,4,4] "Power dissipated in each area of each
        core mapped on the bottom surface volumes";
    Types.Power Qtot "Total dissipated power";
  algorithm
  //Switching sequence
    when sample(0, Ts) then
      for i in 1:4 loop    // core loop
        for j in 1:4 loop  // area loop
          a_idx[j] := mod((a_idx[j]),32) + 1;
          Qv[i,j] := Pvmax*AS[a_idx[j]]*CS[c_idx[i]];
        end for;
      end for;
      if mod(pre(cs_ctr),32) == 0 then
        for i in 1:4 loop
          c_idx[i] := mod((c_idx[i]),32) + 1;
        end for;
      end if;
      cs_ctr := cs_ctr+1;
    end when;
  equation
    for i in 1:4 loop  // core loop
      for j in 1:4 loop     // area loop
        Qvb[:,
            1:(TLC[i,2]+TLA[j,2])*Nr,
            i,j] =
              zeros(12*Nr,(TLC[i,2]+TLA[j,2])*Nr);
        Qvb[1:(TLC[i,1]+TLA[j,1])*Nr,
            (TLC[i,2]+TLA[j,2])*Nr+1:(TLC[i,2]+TLA[j,2]+2)*Nr,
            i,j] = zeros((TLC[i,1]+TLA[j,1])*Nr,2*Nr);
        Qvb[(TLC[i,1]+TLA[j,1])*Nr+1:(TLC[i,1]+TLA[j,1]+2)*Nr,
            (TLC[i,2]+TLA[j,2])*Nr+1:(TLC[i,2]+TLA[j,2]+2)*Nr,
            i,j] = fill(Qv[i,j],2*Nr,2*Nr);
        Qvb[(TLC[i,1]+TLA[j,1]+2)*Nr+1:12*Nr,
            (TLC[i,2]+TLA[j,2])*Nr+1:(TLC[i,2]+TLA[j,2]+2)*Nr,
            i,j] = zeros(10-(TLC[i,1]+TLA[j,1])*Nr,2*Nr);
        Qvb[:,
            (TLC[i,2]+TLA[j,2]+2)*Nr+1:end,
            i,j] = zeros(12*Nr,10-(TLC[i,2]+TLA[j,2])*Nr);
      end for;
    end for;
    for i in 1:N loop
      for j in 1:M loop
        Qb[i,j] = sum(sum(Qvb[i,j,k,l] for k in 1:4) for l in 1:4);
      end for;
    end for;
    Qtot = sum(sum(Qb));
  annotation(
      experiment(StartTime = 0, StopTime = 4, Tolerance = 1e-6,
          Interval = 0.000002),
      __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "euler"));
  end ThermalChip4Cores;
```

```

    model ThermalChipSimpleBoundary "Thermal chip model written by
        explicit ODEs, constant power on half of the bottom surface"
      extends BaseThermalChip;
      parameter Types.Power Ptot = 100 "Total power consumption";
      final parameter Types.Power Pv = Ptot / (N * M / 2) "Power
          dissipated in a single volume";
    equation
      for j in 1:N loop
       for i in 1:div(M, 2) loop
  Qb[j, i] = Pv;
       end for;
      end for;
      for j in 1:N loop
       for i in div(M,2)+1:M loop
  Qb[j, i] = 0.0;
       end for;
      end for;
      annotation(
        experiment(StartTime = 0, StopTime = 0.01, Tolerance = 1e-6,
            Interval = 0.000002),
        __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "euler"));
    end ThermalChipSimpleBoundary;
  end Models;

  package Benchmarks
  end Benchmarks;
end ThermalChipODE;
```

A.2 ThermalChipOO.mo

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

  package Interfaces
    connector HeatPort
      Types.Temperature T;
      flow Types.Power Q;
    end HeatPort;
  end Interfaces;

  package Models

    model Volume
      parameter Types.ThermalConductivity lambda = 148 "Thermal
          conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon"
```

```
        annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation(
        Evaluate = true);
      parameter Types.Temperature Tstart = 273.15 + 40;
      parameter Types.ThermalCapacitance C "Thermal capacitance of a volume";
      parameter Types.ThermalConductance Gx "Thermal conductance of half a volume,x direction";
      parameter Types.ThermalConductance Gy "Thermal conductance of half a volume,y direction";
      parameter Types.ThermalConductance Gz "Thermal conductance of half a volume,z direction";

      Interfaces.HeatPort upper "Upper surface thermal port";
      Interfaces.HeatPort lower "Lower surface thermal port";
      Interfaces.HeatPort left "Left surface thermal port";
      Interfaces.HeatPort right "Right surface thermal port";
      Interfaces.HeatPort top "Top surface thermal port";
      Interfaces.HeatPort bottom "Bottom surface thermal port";
      Interfaces.HeatPort center "Volume center thermal port";

      Types.Temperature T "Volume temperature";
    equation
      C*der(T) = upper.Q + lower.Q + left.Q + right.Q + top.Q + bottom.Q + center.Q;

      upper.Q  = Gx*(upper.T  - T);
      lower.Q  = Gx*(lower.T  - T);
      left.Q   = Gy*(left.T   - T);
      right.Q  = Gy*(right.T  - T);
      top.Q    = Gz*(top.T    - T);
      bottom.Q = Gz*(bottom.T - T);
      center.T = T;
    end Volume;

    model TemperatureSource
      Interfaces.HeatPort port;
      parameter Types.Temperature T = 298.15 "Source temperature";
    equation
  port.T = T;
    end TemperatureSource;

    model PowerSource
      Interfaces.HeatPort port;
      parameter Types.Power Q = 0 "Source thermal power leaving the port";
    equation
      port.Q = -Q;
    end PowerSource;
    partial model BaseThermalChip
      parameter Integer N = 21 "Number of volumesin the x direction";
      parameter Integer M = 21 "Number of volumesin the y direction";
      parameter Integer P = 21 "Number of volumesin the z direction";
      parameter Types.Length L = 12e-3 "Chip lengthin the x direction" annotation(
        Evaluate = true);
      parameter Types.Length W = 12e-3 "Chip widthin the y direction"
        annotation(
        Evaluate = true);
      parameter Types.Length H = 4e-3 "Chip heightin the z direction"
        annotation(
        Evaluate = true);
      parameter Types.ThermalConductivity lambda = 148 "Thermal
        conductivity of silicon" annotation(
        Evaluate = true);
      parameter Types.Density rho = 2329 "Density of silicon"
        annotation(
        Evaluate = true);
      parameter Types.SpecificHeatCapacity c = 700 "Specific heat
        capacity of silicon" annotation(
        Evaluate = true);
      parameter Types.Temperature Tstart = 273.15 + 40;
      final parameter Types.Length l = L / N "Chip lengthin the x
        direction";
      final parameter Types.Length w = W / M "Chip widthin the y
        direction";
      final parameter Types.Length h = H / P "Chip heightin the z
        direction";
      parameter Types.Temperature Tt = 273.15 + 40 "Prescribed
        temperature of the top surface" annotation(
        Evaluate = true);
      final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal
           capacitance of a volume";
      final parameter Types.ThermalConductance Gx = lambda*w*h/l "
        Thermal conductance of a volume,x direction";
      final parameter Types.ThermalConductance Gy = lambda*l*h/w "
        Thermal conductance of a volume,y direction";
      final parameter Types.ThermalConductance Gz = lambda*l*w/h "
        Thermal conductance of a volume,z direction";

      Volume vol[N,M,P](each T(start = Tstart, fixed = true),
                        each C = C,
                        each Gx = 2*Gx, each Gy = 2*Gy, each Gz = 2*Gz)
                            ;
      TemperatureSource[N,M] Tsource(each T(start = Tt));
    equation

      // Connections in the z direction
      for i in 1:N loop
        for j in 1:M loop
        connect(vol[i,j,1].top, Tsource[i,j].port);
        for k in 1:P-1 loop
          connect(vol[i,j,k].bottom, vol[i,j,k+1].top);
        end for;
        end for;
      end for;

      // Connections in the y direction
      for i in 1:N loop
        for k in 1:P loop
        for j in 1:M-1 loop
          connect(vol[i,j,k].right, vol[i,j+1,k].left);
        end for;
        end for;
      end for;

      // Connections in the x direction
      for j in 1:M loop
        for k in 1:P loop
           for i in 1:N-1 loop
             connect(vol[i,j,k].lower, vol[i+1,j,k].upper);
           end for;
         end for;
       end for;
    end BaseThermalChip;

    model ThermalChip4Cores "Thermal chip model written as ODE with 4
        simulated cores"
      extends BaseThermalChip(
        final N = 12*Nr, final M = 12*Nr, final P = 4*Pr);
      parameter Types.Time Ts = 10e-3 "Switching base period";
      parameter Types.Power Ptot = 100 "Total average power consumption
          ";
      final parameter Types.Power Pc = Ptot/4 "Average power dissipated
           by each core";
      final parameter Types.Power Pa = Pc/4 "Average power dissipated
        by each area in a core";
      final parameter Types.Power Pv=Pa/(4*Nr^2) "Average power
        dissipated in a single volume";
      final parameter Types.Power Pvmax = Pv/(sum(AS)/32*sum(CS)/32) "
        Max power dissipated in a single volume";
      parameter Integer Nr=1 "Grid refining factor on x-y plane";
      parameter Integer Pr=1 "Grid refining factor on z direction";
      parameter Integer TLC[4,2] =
        {{1,1},
         {1,7},
         {7,1},
         {7,7}
        } "Upper-left coordinate of each core on base grid";
      parameter Integer TLA[4,2] =
        {{0,0},
         {2,0},
         {0,2},
         {2,2}
        }
      "Upper left coordinate of each area on base grid, relative to
        core";
      parameter Integer AS[32] =
        {1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1,
             1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1}
        "Single area switching sequence"
        annotation(Evaluate=true);
      parameter Integer CS[32] =
        {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1,
             0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0}
        "Core switching sequence"
        annotation(Evaluate = true);
      Integer a_idx[4](start = {1, 9, 17, 25}, each fixed = true) "
        Sequence index of each area";
      Integer c_idx[4](start = {1, 9, 17, 25}, each fixed = true) "
        Sequence index of each core";
      Integer cs_ctr(start = 1, fixed = true) "Core switching counter";
```

```
      Types.Power Qv[4,4] "Power dissipated in each area of each core, per single volume";
      PowerSource Qsource[4,4,2,2] "Array of power sources, first two indeces are core and area";
    algorithm
    //Switching sequence
      when sample(0, Ts) then
        for i in 1:4 loop    // core loop
          for j in 1:4 loop  // area loop
            a_idx[j] := mod((a_idx[j]),32) + 1;
            Qv[i,j] := Pvmax*AS[a_idx[j]]*CS[c_idx[i]];
          end for;
        end for;
        if mod(pre(cs_ctr),32) == 0 then
          for i in 1:4 loop
            c_idx[i] := mod((c_idx[i]),32) + 1;
          end for;
        end if;
        cs_ctr := cs_ctr+1;
      end when;
    equation
      for i in 1:4 loop  // core loop
        for j in 1:4 loop     // area loop
          Qsource[i,j,:,:].Q = fill(Qv[i,j], 2*Nr, 2*Nr);
          connect(Qsource[i,j,:,:].port,
                  vol[(TLC[i,1]+TLA[j,1])*Nr+1:(TLC[i,1]+TLA[j,1]+2)*Nr,
                      (TLC[i,2]+TLA[j,2])*Nr+1:(TLC[i,2]+TLA[j,2]+2)*Nr, P].center);
        end for;
      end for;
    annotation(
        experiment(StartTime = 0, StopTime = 4, Tolerance = 1e-6, Interval = 0.001),
        __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "euler"));
    end ThermalChip4Cores;

    model ThermalChipSimpleBoundary "Thermal chip model written by explicit ODEs, constant power on half of the bottom surface"
      extends BaseThermalChip;
      parameter Types.Power Ptot = 100 "Total power consumption";
      final parameter Types.Power Pv = Ptot / (N * M / 2) "Power dissipated in a single volume";
      PowerSource Qsource[N,div(M,2)](each Q = Pv);
    equation
      connect(Qsource.port, vol[:, 1:div(M,2), P].center);
      annotation(
        experiment(StartTime = 0, StopTime = 0.01, Tolerance = 1e-6, Interval = 0.000002),
        __OpenModelica_simulationFlags(lv = "LOG_STATS", s = "euler"));
    end ThermalChipSimpleBoundary;
  end Models;

  package Benchmarks
  end Benchmarks;
end ThermalChipOO;
```

Bibliography

[1] Llvm project. http://llvm.org/.

[2] Llvm project. http://llvm.org/docs/LangRef.html.

[3] Modelica version 3.3 revision 1 - july 2014, 2014.

[4] gcc. https://gcc.gnu.org, 2020.

[5] Modelica documentation. https://build.openmodelica.org/Documentation/, 2020.

[6] Openmodelica. https://openmodelica.org/, 2020.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology (LTH), 1978.

[9] H. Elmqvist, A. Neumayr, and M. Otter. Modia - dynamic modeling and simulation with julia. 01 2018.

[10] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[12] B. Holger, M. Kurt, S. Guido, and T. Hisao. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 2006.

[13] J. E. Hopcroft and R. M. Karp. A n5/2 algorithm for maximum matchings in bipartite. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971)*, SWAT '71, page 122–125, USA, 1971. IEEE Computer Society.

[14] International Organization for Standardization, editor. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF*. 1996.

[15] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, Nov. 1962.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[17] F. P. Pop A. *MetaModelica: A Unified Equation-Based Semantical and Mathematical Modeling Language.* Lightfoot D.E., Szyperski C. (eds) Modular Programming Languages. JMLC 2006. Lecture Notes in Computer Science, vol 4228. Springer, Berlin, Heidelberg, 2006.

[18] L.-N. Pouchet, A. Größlinger, A. Simbürger, H. Zheng, and T. Grosser. Polly-polyhedral optimization in llvm. In *Polly-polyhedral optimization in LLVM*, volume 2011, 01 2011.

[19] S. Rajopadhye, L. Renganarayana, G. Gupta, and M. Strout. Computations on iteration spaces. In *The Compiler Design Handbook, 2nd ed.*, 2007.

[20] A. Schrijver. *On the history of the transportation and maximum flow problems.* CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, and Department of Mathematics, University of Amsterdam, Plantage Muidergracht 24, 1018 TV Amsterdam, The Netherlands, NL, 2002.

[21] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[22] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, USA, 1983.

[23] P. Zimmermann, J. Fernández, and E. Kofman. Set-based graph methods for fast equation sorting in large dae systems. In *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '19, page 45–54, New York, NY, USA, 2019. Association for Computing Machinery.