Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models

MASSIMO FIORAVANTI, DANIELE CATTANEO, FEDERICO TERRANEO, and SILVANO SEVA, Politecnico di Milano, Italy
STEFANO CHERUBIN, NTNU Norwegian University of Science and Technology
GIOVANNI AGOSTA, FRANCESCO CASELLA, and ALBERTO LEVA, Politecnico di Milano, Italy

Equation-based modelling is a powerful approach to tame the complexity of large-scale simulation problems. Equation-based tools automatically translate models into imperative languages. When confronted with nowadays' problems, however, well assessed model translation techniques exhibit scalability issues that are particularly severe when models contain very large arrays. In fact, such models can be made very compact by enclosing equations into looping constructs, but reflecting the same compactness into the translated imperative code is nontrivial. In this paper, we face this issue by concentrating on a key step of equations-to-code translation, the equation/variable matching. We first show that an efficient translation of models with (large) arrays needs awareness of their presence, by defining a figure of merit to measure how much the looping constructs are preserved along the translation. We then show that the said figure of merit allows to define an optimal array-aware matching, and as our main result, that the so stated optimal array-aware matching problem is NP-complete. As an additional result, we propose a heuristic algorithm capable of performing array-aware matching in polynomial time. The proposed algorithm can be proficiently used by model translator developers in the implementation of efficient tools for large-scale system simulation.

CCS Concepts: • **Computing methodologies** → **Modeling and simulation**; • **Theory of computation** → *Network flows*; *Complexity classes*; • **Software and its engineering** → *Compilers;*

Additional Key Words and Phrases: Equation-based modeling languages, array-aware matching, NP-completeness proof

ACM Reference format:
Massimo Fioravanti, Daniele Cattaneo, Federico Terraneo, Silvano Seva, Stefano Cherubin, Giovanni Agosta, Francesco Casella, and Alberto Leva. 2023. Array-Aware Matching: Taming the Complexity of Large-Scale Simulation Models. *ACM Trans. Math. Softw.* 49, 3, Article 27 (September 2023), 25 pages.
https://doi.org/10.1145/3611661

Authors' addresses: M. Fioravanti, D. Cattaneo, F. Terraneo, S. Seva, G. Agosta, F. Casella, and A. Leva, Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Via Ponzio 34/5, 20133 Milano Italy; emails: massimo.fioravanti@mail.polimi.it, {daniele. cattaneo, federico.terraneo, silvano.seva, giovanni.agosta, francesco.casella, alberto.leva}@polimi.it; S. Cherubin, Department of Computer Science, Norges Teknisk-Naturvitenskapelige Universitet, IT-bygget, Sem Sælands vei 9, 7034 Trondheim; email: stefano.cherubin@ntnu.no.

0098-3500/2023/09-ART27 $15.00
https://doi.org/10.1145/3611661

1 INTRODUCTION

In modern engineering, dynamic modelling and simulation are ubiquitous [16, 64]. Besides providing “virtual prototypes” [52] to streamline plant and control design activities [76, 77], “Digital Twins” – significantly based on simulation [9] – are nowadays the backbone of advanced controls [6, 55], predictive, condition-based and autonomous maintenance [10, 24, 43], anomaly detection, forecast and mitigation [37, 49, 72], continuous integration [54], lifelong asset management [47] and many other applications, see for example the survey in [25]. As a result, unprecedented challenges need facing for rapidly creating, modifying and running simulation models of steadily growing size and complexity [30].

Focusing on simulation models made of **Differential and Algebraic Equations (DAE)**, the *scenario* just sketched has boosted the adoption of declarative, **Equation-Based (EB)** modelling languages as opposite to procedural, imperative programming ones [79]. The key feature of EB languages is the ability of separating the activities of writing a model and of producing its solution algorithms. This ability stems from the fact that the fundamental statement in EB languages is the *equation*. Contrary to the *assignment*, where an *l-value* receives the result of computing the expression on the right hand side, an equation just prescribes that the expressions on the left and the right hand side must be made equal – within convenient tolerances – at every point in time when the solution of a model is computed during its simulation.

Said otherwise, while assignments directly compose the algorithm to compute the model solution, equations just impose constraints to that solution, therefore saying nothing about the solver (in general, numeric) that will be used to compute it. To synthetically express this separation between describing the model and computing its solution, EB models are called *declarative*.

In synthesis, then, EB languages relieve the analyst from the task of turning equations into imperative code to perform their numerical integration, significantly helping to tame the complexity and rapid evolution of modern simulation problems [23]. It is the task of a *translator* to automatically turn a declarative model into an equivalent code in some imperative programming language [34], which is then fed to a compiler.

This translator-compiler workflow was devised at the outset of EB languages, with the aim of decoupling the generation of imperative code (translation) from its optimised compilation into machine code—a task for which e.g., C compilers are very well suited. However, today’s modelling and simulation problems exhibit new characteristics, that require to re-discuss the above translation workflow. A prominent such characteristic, on which we focus in this paper, is the presence of large (and possibly multi-dimensional) array variables and equations. This feature is distinctive of “large-scale” models. Think for example of a 3D thermal model for a solid body with fine-grained spatial discretisation: the model will contain energy dynamic balance equations for each of the many subvolumes into which the solid will be partitioned, and these equations will account for thermal exchanges with the adjacent volumes. It is quite natural to write such a model compactly in EB form by defining suitable array variables and equations, in the latter case by means of looping constructs (examples follow starting from Section 4).

In such cases, as we will show, a trade-off is easily observed. Writing the model directly as imperative code is far more complex, error prone and hard to maintain than adopting an EB declarative framework and obtaining the imperative code by automatic translation. But on the other hand, the code obtained by automatic EB-to-imperative translation is significantly less efficient than the one manually written as imperative.

We argue that the origin of this inefficiency mainly resides in the way EB translators manage array variables and equations. Current production-grade EB translators just treat each component of an array variable or equation as an individual scalar one, which results in a loss of structural information that imperative language compilers cannot efficiently recover [7]. As such, we argue that to improve both the translation time and the efficiency of the produced imperative code, it is necessary to make the translation process “array-aware”.

In this paper we offer a contribution to this end, aiming *both* for an efficient translation and an efficient imperative code. In detail,

(1) we define a figure of merit to quantify how much the looping constructs that make an equation model compact carry over to its imperative translation; building on this figure of merit, we consequently define as *optimal* an array-aware matching that maximally preserves the said looping constructs;
(2) we prove that the optimal array-aware matching problem is NP-complete – contrary to scalar matching, which can be solved in polynomial time and where no such optimality makes sense;
(3) we propose a heuristic algorithm to approximate optimal array-aware matching in polynomial time.

To the best of our knowledge, we are the first to introduce an idea of matching optimality tied to the efficiency of the obtained imperative code, as well as to propose a heuristics that aims for that optimality besides for a fast matching process.

Organisation of the paper. The rest of the paper is organised as follows. In Section 2 we briefly introduce some definitions and the architecture of state-of-the-art approaches. Then, we discuss the history of EB model translators and other array-aware approaches in Section 3. We delineate the scope and purpose of our contribution in Section 4. In Section 5, we define *array-aware matching* and its optimality metric, while we prove its NP-completeness in Section 6. Finally, in Section 7, we show an approximate algorithm for array-aware matching, and in Section 8 we draw some conclusions and highlight future research directions.

2 BACKGROUND ON MODEL TRANSLATION

In this section we outline the foundations of automatic EB-to-imperative translation and compilation. To avoid confusion between scalar and array problems, we first provide a few definitions.

Definition 2.1 (Scalar Variable). A *scalar variable* is an instance of a system property, identified by a name, whose value at every instant is fully defined by a scalar number and – possibly – a unit of measurement.

Definition 2.2 (Array Variable). An *array variable* is a collection, identified by a name, of one or more scalar variables, each one referenceable through one or more integer indices.

Definition 2.3 (Array Equation). An *array equation* is a collection of one or more scalar equations, expressed compactly as a single parametric equation that references by index one or more scalar components of one or more array variables.

Definition 2.4 (Array Dimensionality and Size Vector). The *dimensionality* of an array is the number of dimensions of that array, that is, the number of integer indices needed to reference a single scalar component in it. We assume by convention that the said indices are 1-based. Their maximum values collectively form the array *size vector.*

It follows that, in EB modelling languages, *array variable*s are akin to the concept of ordinary multidimensional arrays, commonly found in many programming languages. Also, in EB modelling languages, an *array equation* is obtained by encasing a scalar one in one or more nested looping constructs, that define the indices of the contained scalar variables within the arrays of which they are part, as well as the ranges for the said indices.

It is important to highlight that the mentioned looping constructs – differently from those of imperative programming languages – allows the user to predicate on array equations, such as

$$x[i] = y[i] \ \forall i \in [1,3]; \tag{1}$$

to be intended as an abbreviation of

$$x[1] = y[1];\ x[2] = y[2];\ x[3] = y[3]; \tag{2}$$

This noted, the translation-compilation process – as per the current state of the art in both research and production tools, see e.g., [23, 33] and [1, 2, 4, 5], respectively – can be divided into the following steps.

Flattening. The model equations, independently of the way they were input by the user – e.g., as a single set, by hierarchically instantiating and interconnecting subsystems, or anyhow else – are brought to be one set of scalar DAEs. This step contains a sub-step named *loop unrolling*, in which each expression in the form shown in Equation (1) is replaced by its set of scalar components, as shown in Equation (2). The outcome of flattening is thus a DAE system with scalar equations and variables.

Matching. This step (hereinafter denoted as *scalar* matching when confusion may occur) consists of coupling each (scalar) equation to one (scalar) variable, meaning that the equation is the initial candidate for computing the variable at simulation time. A failure in matching indicates a model inconsistency (e.g., and most typically, an equations/variables imbalance). In the general case, matching may also require an index reduction sub-process, implemented by methods such as the Pantelides algorithm [58]. Our paper does not address index reduction.

Scheduling. This step determines the order in which the equations are solved. The (scalar) equations of the system are ordered accordingly to their mutual dependencies, as established by the matching process. For example, the equation matched with variable v – that is, the candidate one to compute v – is scheduled before all other equations in which v appears. The ideal result would allow to compute the solution variable by variable, in sequence. This is hardly ever obtained, however. During the scheduling step, some cyclic dependency among variables may arise, and as a result the one-to-one relationship established by the matching process between those variables and their candidate equations cannot be maintained. Cyclic dependencies indicate the existence of a so-called ***Strongly Connected Component*** **(SCC)**: the involved variables will need computing all together, most often numerically. The Tarjan algorithm [73] is a commonly used means to determine the equation solution order and to identify SCCs.

Code generation. The last step is the generation of imperative simulation code for a specific choice of the numerical integration algorithm. The simulation code can be self-contained for certain numerical integration algorithms, such as explicit ones, or can rely on external solver libraries, such as those in the SUNDIALS suite [38].

After the translation and compilation process is complete, the obtained executable code is run to produce the simulation output, in the form of a table with the value of the model variables as a function of simulation time. This process is often part of a graphical modelling environment for rapid prototyping. Once the modeller decides to perform a simulation, all the translation, compilation and execution steps are on the critical path toward getting the simulation results. Consequently, the shortening of compilation and simulation times is especially important for this kind of use-cases.

3 RELATED WORK

In this section we spend some words to relate our proposal to the research *scenario* on EB modelling in general, and to neighbouring research on EB-to-imperative translation in particular.

In the landscape of simulation languages, EB ones appeared and gained visibility in the 80s/90s of the last century; notable examples are Omola [11], Dymola [28] and gPROMS [15]. A taxonomy would stray from the scope of this paper; the reader interested in the historical *panorama* can refer e.g., to [22]. Worth noticing, however, is the common ancestor for the boost of the declarative approach stemming from studies such as [13] and [29], where the idea that simulation languages had to abandon the imperative setting was set forth and preliminarily exploited. Research on the matter thus focused at first on the model manipulation [41, 50] required by going declarative, and after a long systematisation process, this resulted in the birth of the Modelica language [51] to which we refer herein (though all the ideas we propose are general to the EB context).

The engineering use of EB languages and tools sustainably spread out in various domains, ranging from the chemical [12] and process industry [35] to power generation [21] and transmission [71], mechatronics [75] and robotics [39], automotive [42], and vehicles at large [26], aerospace [53], buildings [68] and more, including control design [20] and diagnostics [17]; the papers in the necessarily limited list above also contain interesting bibliographies for the reader willing to investigate further.

Together with testifying the success of the EB approach, however, the expansion just mentioned also shed light on some relevant limitations of the existing EB tools [62] – not of EB languages by themselves, it is worth stressing – especially when dealing with large-size models [19, 46]. This was the motivation for a first wave of tool optimisation, having as a major point the introduction of sparse solvers, a well-treated and long-lasting matter in domain-specific tools – see e.g., [32, 45, 69, 70] – but a source of challenges in the inherently multi-domain EB one [57, 78]. Examples of this research – with specific reference to Modelica given our scope – are [18, 63].

The possibility of *solving* large models fast enough to widen the EB applicability perimeter evidenced however a second type of tool limitation, concerning the *translation* rather than the solution of such models [66, 67]. The matter became critical in recent years, together with the emergence of problems that require model-based prototyping [48] and can scale up to the order of 10^5 equations. When such models become part of the inherently iterative engineering process, the time spent in translating and compiling them can be comparable to that spent in running simulations, if not even dominant [14]. For the sake of clarity it is worth noticing that the million equation barrier was already approached in the past [74] and in some domains nowadays well trespassed [61] by simulation tools, but these tools *are not of the EB type*, and most notably, do not separate model description and solution—which is a primary goal of the EB approach.

As a result, EB tools are nowadays undergoing a second wave of optimisation, directed to efficient translation. Open-source parsers [59] for EB languages such as Modelica are available providing some degree of array preservation, thereby enabling the research community to experiment with making the translation pipeline array-aware. In this relatively new effort, a primary objective is to achieve an O(1) scaling of the translation time with the size of the model arrays, that as already noted are the main cause for the inefficiency of scalar-only model manipulation. In this context, the nearest neighbouring work to our research is the paper by Zimmermann et al. [80], who introduce the concept of "set-based graph" as a means to re-state the matching problem (originally scalar) in such a way to achieve an O(1) translation, together with proposing algorithms for other manipulation steps related to matching, such as the management of strongly connected components and scheduling.

The main difference of our research with respect to [80] is a twofold instead of a single goal. More precisely, we do not aim just for an efficient translation, but also for an efficient simulation code.

If the efficiency of the simulation code is taken into account, the number of looping constructs in the EB model that are preserved in the imperative one comes to matter a lot. The set-based approach of [80] is not designed to take this aspect into account. Aiming at loop preservation straightforwardly entails the introduction of an idea of optimality. This moves the focus from array-aware matching to *optimal* array-aware matching, and owing to its NP completeness, to the need for heuristics.

Other works have addressed the array-aware matching problem, such as [65] which correctly noticed that preserving looping constructs can positively impact both translation and simulation time. They also present a prototype translator that is limited to handling systems without algebraic loops and resorts to flattening the model completely in cases where their array-aware matching algorithm does not produce a solution. The paper [56] addresses array-aware index reduction, and the corresponding Modia implementation [3] also includes an array-aware matching algorithm that however is very simple and cannot split variable and equation nodes, thus requiring the modeller to pre-process the input model so as to make sure that every array variable can be matched to exactly one array equation. Compared to the two previously quoted works, our paper introduces the concept of optimal array-aware matching and proves that achieving such optimality is an NP complete problem, as well as presenting a more complete array-aware matching algorithm.

4 RESEARCH MOTIVATION

In this section, we discuss the inefficiencies that arise from a non array-aware model translation. To ground the discussion on an example, consider the model of a thermally insulated metal wire with prescribed temperatures at its ends. The evolution of this system is ruled by the one-dimensional Fourier equation. Carrying out a uniform spatial discretisation with the finite-volume approach results in the following system of differential equations:

$$c\dot{T}_i = \begin{cases} g(2T_{left} - 3T_i + T_{i+1}) & i = 1 \\ g(T_{i-1} - 2T_i + T_{i+1}) & \forall\, i \in [2, N-1] \\ g(T_{i-1} - 3T_i + 2T_{right}) & i = N \end{cases} \tag{3}$$

where $N > 3$ is the number of finite volumes, c is the thermal capacity of a volume, g the thermal conductance between the centres of two adjacent volumes, T_i is the temperature of volume i, with T_1 being the leftmost and T_N being the rightmost volume, and finally T_{left} and T_{right} are the prescribed side temperatures.

When expressed in Modelica, the wire model reads as follows.

```
model Thermal1D
    parameter Integer N      = 5;
    parameter Real    g      = 0.00314785;   // W/K
    parameter Real    c      = 0.2707936;    // J/K
    parameter Real    Tleft  = 400 + 273.15; // K
    parameter Real    Tright = 20 + 273.15;  // K
    Real T[N](each start=Tright);                   // array variable

equation
    c*der(T[1])  = g*(2*Tleft - 3*T[1] + T[2]);
    for i in 2:N-1 loop                             // looping construct to express
        c*der(T[i]) = g*(T[i-1] - 2*T[i] + T[i+1]); // an array equation
    end for;
    c*der(T[N]) = g*(T[N-1] - 3*T[N] + 2*Tright);
end Thermal1D;
```

Listing 1. The model of a wire, as shown in Equation (3), as expressed in Modelica code.

The remarkable similarity between the Modelica model and original system of equations is apparent; the *der* operator is used for expressing the derivative with time.

It is evident that the availability of array equations make EB models assume a compact and easily readable form. However, present state-of-the-art translators provide looping constructs *only* as a convenience for the modeller, and do not take advantage of this structural information to improve the translation efficiency, nor that of the generated simulation code. As a consequence, some inefficiencies arise that would be completely unexpected in the world of imperative programming languages.

A first such inefficiency is that the amount of both time and memory needed for the translation scale superlinearly with the size of array equations, rather than exhibiting the expected O(1) scaling typical of imperative languages. In fact, the compilation of imperative programming languages treats looping constructs explicitly. The instructions inside a loop are represented in the compiler data structures just once, irrespective of the loop iteration count, and all subsequent compilation steps are performed on this compact data structure.

On the contrary, state-of-the-art EB translation algorithms are designed to only work in terms of scalar equations (see *Flattening* in Section 2), which are stored in the translator data structures individually. These scalar equations are then passed on to subsequent translation steps, some of which scale superlinearly. It is not uncommon for models exceeding the 10^5 equations mark to require translation times in the order of hours, and working memory in the order of hundreds of gigabytes [14].

A second and consequent inefficiency is that the size of the produced imperative code scales linearly with the size of array variables and equations. Since the original EB model is scalarised, the simulation consists of procedures containing repetitive code instead of looping constructs, often amounting to several gigabytes for large-scale models. Although imperative compilers are very efficient and can achieve O(1) scaling in compilation with respect to array sizes, this is only possible if they are given a source code with loops, not long lists of repetitive statements.

The last inefficiency regards the performance of the machine code produced by the compiler when fed with the automatically generated imperative code. Modern computer architectures are built upon assumptions such as the *locality principle* for both data and code, which are the theoretical foundations for caches [27] and other microarchitectural optimisations. However, the code produced by current-generation model translators is not able to exploit these optimisations. First, the execution of large blocks of straight-line code requires frequent *instruction* cache invalidation. Furthermore, the loss of structural information about arrays leads to the generation of code that exhibits irregular data access patterns. Therefore, repetitive machine code is not only large, but runs significantly slower than equivalent hand-written code, in some cases 100 times or more [8].

Both translators and compilers can in principle infer looping constructs and improve access locality, but their ability to perform this kind of optimisation is limited [44]. Additionally, such inferences require to repetitively scan the list of flattened statements, making O(1) scaling apparently impossible. We thus argue that a far better approach would be to preserve array-awareness throughout the translation process, rather than try to regain it *a posteriori*.

Summing up, all the presented inefficiencies share their root cause in the fact that existing model translation algorithms are not array-aware. In this paper, we begin an effort to fill this gap. Since the flattening step is trivial to extend for array-awareness – it suffices to not unroll looping constructs – we focus on the second and first truly key step, i.e., the matching problem.

5 GRAPH REPRESENTATION AND PROBLEM STATEMENT

In this section, we introduce the notation to describe array-aware algorithms, and present the formal statement of the array-aware matching, the algorithm we focus on in this paper.

We denote with $G = (N, D)$ a generic graph having set of nodes N and set of arcs D. Any set of scalar equations E in the scalar variables V can be represented with a bipartite graph in which

- $N = V \cup E$,
- $V \cap E = \emptyset$,
- $D \subseteq V \times E$,
- the presence of an arc (v_i, e_j) indicates that variable v_i appears in equation e_j.

In this context, the set of arcs D represents dependencies between variable and equation nodes.

To include array variables and equations – including multidimensional ones – we extend the above notation as follows. We indicate with $\underline{v}_i$ the generic multidimensional array variable, whose scalar components in the physical problems we target are invariantly real numbers. We denote by ζ_i the dimensionality of $\underline{v}_i$. We denote with $K = \{k_1, \dots, k_\delta, \dots, k_{\zeta_i}\}$ the sequence of positive integers needed to reference a scalar component within the array variable $\underline{v}_i$. Each generic index k_δ ranges from 1 to the size of the corresponding dimension δ, with $1 \leq \delta \leq \zeta_i$. For compactness, we synthetically write $v_{i,K}$ to indicate the scalar component in $\underline{v}_i$ of indices $\{k_1 \dots k_{\zeta_i}\}$.

Consistently with the above notation, we define the following.

- $\underline{V}$ is the set of array variables $\{\underline{v}_i\}$, $i = 1, \dots, |\underline{V}|$ in the model; if the model contains scalar variables, these are considered array variables of unitary dimensionality and size;
- $\underline{E}$ is the set of array equations $\{\underline{e}_j\}$, $j = 1, \dots, |\underline{E}|$ in the model; if the model contains scalar equations, they are considered array equations of unitary dimensionality and size.

A model containing both scalar and array variables and equations can thus be represented with any of the two equivalent bipartite graphs defined as follows.

(1) The first one is obtained by just setting $N = \{v_{i,K}\} \cup \{e_{j,L}\}$, and $D \subseteq V \times E$ as the scalar dependencies. The presence of an arc $(v_{i,K}, e_{j,L})$ indicates that the scalar variable $v_{i,K}$ appears in the scalar equation $e_{j,L}$. Observe that in the topology of such a graph any information concerning the existence of array variables and equations is lost. We name this the *flattened* graph.

(2) The second one is obtained by setting $\underline{N} = \underline{V} \cup \underline{E}$, and $\underline{D} \subseteq \underline{V} \times \underline{E}$ as the array dependencies. In this case the presence of an arc $(\underline{v}_i, \underline{e}_j)$ indicates that at least one scalar variable in $\underline{v}_i$ appears in at least one scalar equation in $\underline{e}_j$. We name this the *array* graph.

The above definitions imply that the array graph is homomorphic to the flattened one. However, for the two graphs to be equivalent, each arc in the array one needs to carry information about which components of the connected array variable and equation it refers to — a matter that does not pertain to the scalar case. Therefore, the arcs of the array graph must be endowed with the information needed to reconstruct the arcs of the flattened graph. To formalise this, we introduce the concept of *local multidimensional incidence matrix*. Given an array equation $\underline{e}_i$ and an array variable $\underline{v}_j$ of dimensionality ζ_i and ζ_j respectively, let $K_i = \{k_{i,1}, \dots k_{i,\zeta_i}\}$ be the sequence of indices for $\underline{e}_i$, and $L_j = \{l_{j,1}, \dots l_{j,\zeta_j}\}$ be the sequence of indices for $\underline{v}_j$. The multidimensional local incidence matrix $u_{i,j}$ has dimensionality $\zeta_{ij} = \zeta_i + \zeta_j$, and its sequence of indices Q_{ij} is the concatenation of K_i and L_j. Its generic element is 1 iff the scalar variable $v_{j,K}$ appears in the scalar equation $e_{i,L}$, else it is 0. We name U the set of local multidimensional incidence matrices. An example of a Modelica model and the corresponding array graph can be found in Figure 1.

The notation we just provided is capable of representing arbitrary multidimensional incidence matrices. However, it should be noted that those produced by EB models coming from equations of physics are significantly structured, and present patterns that arise out of the looping constructs and expressions used for accessing array variables within array equations. It follows that, although

```
c*der(T[1])  = g*(2*Tleft - 3*T[1] + T[2]);     // e1
for i in 2:N-1 loop
    c*der(T[i]) = g*(T[i-1] - 2*T[i] + T[i+1]); // e2
end for;
c*der(T[N]) = g*(T[N-1] - 3*T[N] + 2*Tright);   // e3
```

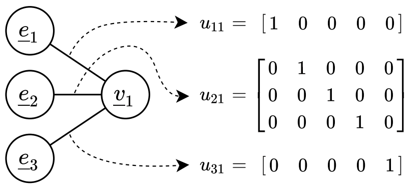
$\underline{e}_1$
$\underline{e}_2$
$\underline{e}_3$
$\underline{v}_1$
$u_{11} = [1\ 0\ 0\ 0\ 0]$
$u_{21} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$
$u_{31} = [0\ 0\ 0\ 0\ 1]$

Fig. 1. The array equations of the Modelica model of a wire previously shown in Listing 1, and its corresponding array graph. Variable $\underline{v}_1$ is $\dot{T}$, the only non-state array variable in the model. Thus, the columns of the u matrices correspond to $\dot{T}_1$, $\dot{T}_2$, $\dot{T}_3$ and so on.

from a theoretical standpoint it is convenient to reason in terms of multidimensional incidence matrices, an industry-grade implementation would rely on an efficient pattern-based data structure to achieve O(1) scaling. This important matter is discussed in Section 7.4.

5.1 The Array-aware Matching Problem

The array-aware matching is an operation that takes as input an array graph $\underline{G} = (\underline{V} \cup \underline{E}, \underline{D})$, where we recall that $\underline{D}$ is the set of dependencies between array variables $\underline{V}$ and array equations $\underline{E}$. Additionally, every dependence $\underline{D}$ has an associated local multidimensional incidence matrix $u_{i,j}$. The array-aware matching produces as output an array graph $\underline{G}' = (\underline{V} \cup \underline{E}, \underline{D}')$ with the following properties.

(1) $\underline{D}' \subseteq \underline{D}$
(2) Each arc $(\underline{v}_i, \underline{e}_j) \in \underline{D}'$ has an associated local multidimensional incidence matrix $m_{i,j}$, where $m_{i,j}(\underline{v}_{i,K}, \underline{e}_{j,L}) \in \{0, 1\}$. We call M the set of $m_{i,j}$.
(3) $\forall m_{i,j} \in M$, $m_{i,j}$ has the same size and dimensionality of $u_{i,j} \in U$
(4) $\nexists (\underline{v}_i, \underline{e}_j) \in \underline{D}' : m_{i,j}$ is a zero matrix
(5) $u_{i,j}(\underline{v}_{i,K}, \underline{e}_{j,L}) = 0 \implies m_{i,j}(\underline{v}_{i,K}, \underline{e}_{j,L}) = 0$
(6) $\forall \underline{v}_{i,K} \in \underline{V} : \exists! m_{i,j}(\underline{v}_{i,K}, \underline{e}_{j,L}) = 1$

Property (1) states that no new arcs are added to the graph, (2–3) denote as "matching matrices" m the local incidence matrices of the output graph, (4) states that – consistently – arcs fully unmatched are removed, (5) ensures that a scalar equation is matched only with a scalar variable it contains, and (6) that each scalar equation is matched to one and only one scalar variable.

Definition 5.1 (Optimal Array-aware Matching). Let us define the *expansion function* $f_e(m)$, which maps a local incidence matrix m to the number of looping constructs required to implement the matching it expresses. $f_e(m)$ has a value of zero if and only if m is a zero matrix, otherwise $f_e(m) \geq 1$. We say that a matching $(\underline{G}', M)$ is optimal if it minimizes the following:

$$\Omega(\underline{G}, M) = \sum_{m \in M} f_e(m) \tag{4}$$

For the purpose of this work, it is not necessary to fully specify the expansion function, because different model translators may be able to "efficiently" handle only a subset of all possible matching matrices. In this discussion, the metric of "efficiency" is therefore the ability to represent the matching described by $m_{i,j}$ as a single looping construct consisting of the scalar equations of the node adjacent to the corresponding arc $d_{i,j}$. Without loss of generality we can consider the ideal case in which every matching matrix maintains its correspondence with exactly one looping construct. In this case, the f_e function takes this form:

$$f_e(m) = \begin{cases} 0 & \text{iff } m \text{ is a zero matrix} \\ 1 & \text{otherwise} \end{cases} \tag{5}$$

In this case, the optimality metric Ω is equal to the number of arcs where at least one equation is matched. Intuitively, the more arcs that have a matching, the higher the number of times the same array equation has been matched with multiple variables, and for each variable matched with the same array equation a separate looping construct must be introduced. Therefore, minimizing Ω means minimizing the number of looping constructs.

5.1.1 Just Enough, but Not Too Much. It is important to stress that the optimality metric Ω is but a proxy for identifying the matching that best improves both translation and simulation time. With the definition we gave of array equation in this paper (Definition 2.3 in Section 2) where individual scalar equations in an array equation differ only by the array indices, reducing the number of looping constructs does indeed result in more efficient code. Other works however, such as [56] and the corresponding Modia implementation [3] allow if statements in array equations thereby allowing to merge in a single array equation also scalar equations that are structurally different. For example, the wire model written with such an extended definition of array equations would read as

```
for i in 1:N loop
    c*der(T[i]) =      if(i == 1) then g*(2*Tleft - 3*T[1] + T[2])
                  else if(i == N) then g*(T[N-1] - 3*T[N] + 2*Tright)
                  else                 g*(T[i-1] - 2*T[i] + T[i+1]);
end for;
```

Although it would appear that such an array equation could improve the optimality metric by requiring a single looping construct that handles all the wire finite volumes instead of requiring one looping construct plus two scalar equations for the first and last volume, this would *not* result in efficient code. Indeed, if such an array equation were brought as-is till code generation, the imperative code would need if statements in the loop to handle the differences in the equation structure, and the cost at run-time of the if statements would need to be paid multiplied by every loop iteration and additionally multiplied by every simulated time step. This is a strong point in favour of our definition of array equations, that explicitly disallows grouping structurally different scalar equations in a single array equation. Moreover, to achieve the best efficiency, a translator would need to perform symbolic manipulations to transform inefficient code such as the one above by moving the structurally dissimilar equations outside of the loop before the matching step, should the modeller decide to write the model in that form.

6 COMPLEXITY OF OPTIMAL ARRAY-AWARE MATCHING

In this section we show that for what concerns the matching problem, preserving the array structure of both equations and variables results in NP-completeness. This theoretical result motivates the need for heuristic algorithms, that will be presented in Section 7.

THEOREM 6.1 (COMPLEXITY OF OPTIMAL ARRAY-AWARE MATCHING). *The problem of producing an optimal array-aware matching is NP-complete.*

PROOF. We prove Theorem 6.1 by reducing the max-2-sat problem to optimal array-aware matching. Max-2-sat was proven NP-complete by Garey, Johnson and Stockmeyer [36], and reads as follows: given a Boolean formula in conjunctive normal form, where each clause contains at most two literals, find a literal assignment such that the maximum number of clauses is satisfied.

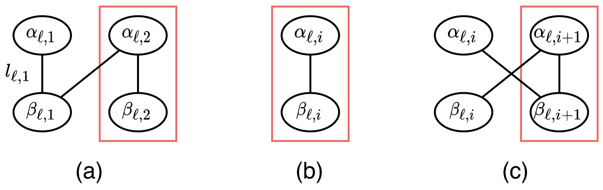

Fig. 2. Subgraphs used by the procedure for building the intermediate graph G from a list of AND clauses.

First of all, we establish a procedure that allows to represent a max-2-sat problem in terms of array-aware matching. To this end we introduce three formal modifications to the way max-2-sat is expressed in order to simplify the reduction process.

(1) Every clause in the form (a) is rewritten as $(a \vee a)$. It is evident that this rewriting does not change the value of that clause for any literal assignment.
(2) Instead of considering the input to max-2-sat as a conjunction of clauses, we consider it as an ordered list. This is legitimate, as maximising the number of satisfied clauses does not require knowing if the entire formula is satisfied.
(3) We assume that, when traversing the list of clauses, the first encountered occurrence of each literal is not negated. Should this be false, one would simply have to replace that literal with another one defined as its complement.

The above said, to encode a max-2-sat instance into an array-aware matching one, we start by recalling that

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

and we replace each OR clause in the list with the above defined equivalent triple of AND ones. Observe that only one of the three AND clauses can be true, hence maximising the number of satisfied AND clauses in the reformulated list is equivalent to the original problem.

Let us start with two empty sets of nodes $A = \emptyset$, $B = \emptyset$. Then, we build an intermediate flattened bipartite graph $G = (A \cup B, L)$ by traversing the list of AND clauses. By construction, we will retain the invariant $A \cap B = \emptyset$. For each clause c we operate as follows.

First, we consider the first literal ℓ in the clause. If this is the first occurrence of ℓ, we add to G the subgraph shown in Figure 2(a). In doing so, α nodes should be considered belonging to the A set while β nodes should be considered part of the B set. We name $e_{\ell,1}$ the first edge associated to literal ℓ. We name $\alpha_{\ell,1}$ and $\beta_{\ell,2}$, respectively the *start* and the *end node* for that literal.

If ℓ was already encountered and appears here in non-negated form, we add to G the subgraph shown in Figure 2(b), and we connect $\alpha_{\ell,i}$ to the end node of the literal. The start node of the literal does not change, while its end node becomes $\beta_{\ell,i}$.

If ℓ was already encountered and appears here in negated form, we add to G the subgraph shown in Figure 2(c), and we connect $\alpha_{\ell,i}$ to the end node of the literal. The start node of the literal does not change, while its end node becomes $\beta_{\ell,i}$.

We repeat the subgraph insertion process for the second literal in the clause.

For each clause c, we define a set of four nodes – taken from the two subgraphs just added – as the union of nodes highlighted in red in Figure 2. We name this set *clause nodes* of c and we name it N_c.

We repeat the above for all clauses. When the end of the list is reached, we connect the end node of each literal with its start one. Doing so, we create for each literal a simple cycle $\alpha_{\ell,1}$, $\beta_{\ell,1}, \ldots, \alpha_{\ell,n_\ell}, \alpha_{\ell,1}$ of even cardinality n_ℓ, ordered as just indicated. We number the edges of each cycle as $l_{\ell,1}$ through l_{ℓ,n_ℓ}. This concludes the construction of graph G.

Graph G is a flattened graph and as such does not have the form required by array-aware matching, thus we need to construct a different graph $\underline{G} = (\underline{E} \cup \underline{V}, \underline{D})$ homomorphic to G, and expressed in terms of array equations and variables. Therefore, $\underline{G}$ will satisfy the following properties:

- For each node $\alpha_{\ell,i} \in A$ not part of a clause node N_c, there exists a variable $\underline{v}_n \in \underline{V}$ with dimensionality 1 and size 1.
- For each node $\beta_{\ell,j} \in B$ not part of a clause node N_c, there exists an equation $\underline{e}_m \in \underline{E}$ with dimensionality 1 and size 1.
- For each clause node N_c there exists a variable $\underline{v}_n \in \underline{V}$ and an equation $\underline{e}_m \in \underline{E}$, both with dimensionality 1 and size 2.
- The local incidence matrix $u_{n,m}$ is a square identity matrix in order to represent the original relationship found in the scalar graph.
- No other variables nor equations exist in $\underline{G}$.
- Dependencies arcs $\underline{D}$ are constructed so as to make $\underline{G}$ homomorphic to G.

In other words, all the α nodes are considered variables, all the β nodes are considered equations, and each labelled set of four nodes N_c forms an array equation of size 2, and a corresponding array variable of size 2, while unlabelled nodes translate to scalars.

To carry on, we now need the following lemma.

Lemma 6.2 (Complexity of the Construction of $\underline{G}$). *The graphs G and $\underline{G}$ can be constructed in polynomial time.*

Proof. The construction process which defines G is linear with the number of clauses and the creation of $\underline{G}$ can be done by simply enumerating the nodes and edges of G. □

Back to the main proof, once we obtained the matched graph by executing array-aware matching on $\underline{G}$, we assign to each literal ℓ the boolean value *true* if arc $l_{\ell,1}$ has been selected for the matching, *false* otherwise. This last step can also be performed in polynomial time.

Given the definition of optimal array-aware matching of Definition 5.1, an algorithm capable of solving the problem will maximise the number of $(l_{\ell_1,i}, l_{\ell_2,j})$ arc pairs between nodes belonging to the clause node sets N_c. In fact, each pair contributes only a unitary weight to the optimality metric Ω, while two non-paired arcs will contribute a weight of two.

By construction of the bipartite graph G, each node, be it an equation or a variable, has exactly two adjacent edges. Thus, the matching choice is binary. Since we have built a simple cycle for each literal, there are only two matching solutions for each cycle, one selecting $l_{\ell,1}$ and all the odd numbered arcs, the other one selecting all the even numbered ones.

Additionally, due to how we constructed each cycle, the arc connecting the nodes belonging to a clause node set N_c for cases (a) and (b) of Figure 2 is odd, while in case (c) it is even. Thus, if the arc in the red box of Figure 2(a) is selected, then all the arcs in boxes of subgraph type (b) will also be selected, the arcs in boxes of subgraph type (c) will not be selected, and *vice versa.*

It follows that an arc pair inside a labelled node set can be selected if and only if the literal assignment – as read from the graph – satisfies the corresponding clause. As such, the objective functions of max-2-sat and array-aware matching, given the proposed graph construction and interpretation, are equivalent. This implies that array-aware matching is NP-complete. □

Example. For the convenience of the reader, we complement the formal proof with an example of how the intermediate bipartite graph can be built from the following list of AND clauses:

$$\{a \wedge b,\ \neg a \wedge c,\ c \wedge d\} \tag{6}$$

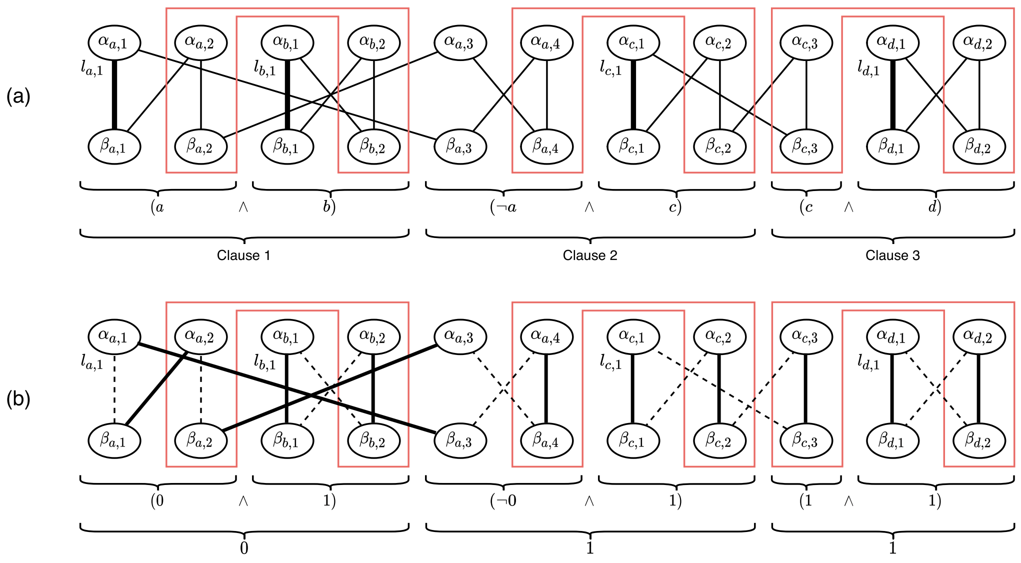
(a)
(b)
$\alpha_{a,1}$ $\alpha_{a,2}$ $\alpha_{b,1}$ $\alpha_{b,2}$ $\alpha_{a,3}$ $\alpha_{a,4}$ $\alpha_{c,1}$ $\alpha_{c,2}$ $\alpha_{c,3}$ $\alpha_{d,1}$ $\alpha_{d,2}$
$l_{a,1}$ $l_{b,1}$ $l_{c,1}$ $l_{d,1}$
$\beta_{a,1}$ $\beta_{a,2}$ $\beta_{b,1}$ $\beta_{b,2}$ $\beta_{a,3}$ $\beta_{a,4}$ $\beta_{c,1}$ $\beta_{c,2}$ $\beta_{c,3}$ $\beta_{d,1}$ $\beta_{d,2}$
$(a \wedge b)$ $(\neg a \wedge c)$ $(c \wedge d)$
Clause 1
Clause 2
Clause 3
$(0 \wedge 1)$ $(\neg 0 \wedge 1)$ $(1 \wedge 1)$
0
1
1

Fig. 3. Bipartite graphs representing the list of AND literals shown in Equation (6). Graph (a) shows the correspondence between each clause and literal and the nodes and arcs in the graph. Graph (b) shows an optimal matching for this graph. Solid lines represent matched arcs, dashed lines represent non-matched arcs.

The list of clauses is scanned left to right. The first clause encountered is $(a \wedge b)$, and both a and b are literals being encountered for the first time. As a result, in the graph we add two structures of type (a) shown in Figure 2. The start node of a is $\alpha_{a,1}$, the end node of a is $\beta_{a,2}$. In the same way, the start node of b is $\alpha_{b,1}$, the end node of b is $\beta_{b,2}$.

Now, the second clause $(\neg a \wedge c)$ is processed. The literal a was already encountered, and appears in negated form; as a result, a structure of type (c) from Figure 2 is inserted, and the $\beta_{a,2}$ end node is connected with $\alpha_{a,3}$. The end node of a is changed to the newly inserted $\beta_{a,3}$. Instead, the literal c is newly encountered, and therefore we insert a structure of type (a) from Figure 2. The start node of c is $\alpha_{c,1}$, and the end node of c is $\beta_{c,2}$.

The process has now arrived at the third and last clause $(c \wedge d)$. The literal c makes a reappearance in positive form, and therefore a structure of type (b) is inserted. The end node $\beta_{c,2}$ is connected to the new node $\alpha_{c,3}$, and $\beta_{c,3}$ becomes the new end node of c. Finally, the literal d is encountered for the first time, and a structure of type (a) is inserted. At this point the list of clauses is exhausted.

The last step for constructing the scalar graph consists in connecting the end node of each literal with its corresponding start node. For literal a, we connect $\alpha_{a,1}$ with $\beta_{a,3}$. For literal b, we connect $\alpha_{b,1}$ with $\beta_{b,2}$. For literal c, we connect $\alpha_{c,1}$ with $\beta_{c,3}$. Finally, for literal d, we connect $\alpha_{d,1}$ with $\beta_{d,2}$.

Figure 3(a) shows the bipartite graph generated through the steps we have just outlined. Additionally, the figure outlines in red each clause node, and highlights the first arc of each cycle, which is used to determine the value of each literal from the matching.

Figure 3(b) highlights the optimal matching of the same graph, alongside with the value of each clause and each literal. Since arc $l_{a,1}$ has not been used in the matching, a is assigned value 0. Instead, literals b, c and d are assigned a value of 1, since the arcs $l_{b,1}$, $l_{c,1}$, and $l_{d,1}$ are all used in the matching. This matching maximises the number of array equations and variables matched, and therefore also minimizes Ω: two arrays are matched, out of the three arrays in the input graph.

Each array corresponds to a clause, and the matched arrays represent clauses whose value is one. In fact, the literal assignment makes the second and third clause true, and the first clause false.

7 ARRAY-AWARE MATCHING ALGORITHM

The NP-completeness proof of optimal matching highlights how, in order to efficiently handle large-scale problems, there is a need to introduce suitable heuristics. In this section, we propose an algorithm of reduced complexity.[1]

Our proposal is a partially heuristic two-step procedure. The first step identifies obligatory matching choices and removes them from the problem, thereby reducing its size. We call this first step, presented in Section 7.2, the *simplification* step. No heuristics are involved in it.

For the second step, that we call the *matching* step, we propose an extension of the Hopcroft-Karp [40] algorithm to array graphs, which allows to exploit local incidence matrices in such a way to preserve the existing looping constructs. The matching step, described in Section 7.3, terminates at the first solution found, whence its heuristic nature.

Before presenting the simplification and matching algorithms, it is necessary to define some operations on local incidence matrices that are used throughout the said algorithms. This is done in Section 7.1, but requires an important preliminary remark. When dealing with matching problems that contain multidimensional arrays, incidence matrices become multidimensional as well. To lighten the treatise, in this work we nonetheless stick to talking about "rows" and "columns". This notation does not cause any generality loss, however. All the proposed algorithms can work in the case of multidimensional incidence matrices by simply interpreting "rows" and "columns" as *sets* of dimensions, referring respectively to equations and variables. We also talk about "row" and "column vectors", that generalize to matrices having only one of the two sets of dimensions.

7.1 Operations on Local Incidence Matrices

Since incidence matrices are boolean, it is trivial to define the logical operators *conjunction* (*and*, $\wedge$), *disjunction* (*or*, $\vee$) as well as *negation* (*not*, $\neg$) on matrices of the same dimensions as the element-wise operations. Additionally, for convenience, we define the operation $a \setminus b$ (*subtraction*) as $a \wedge \neg b$.

We also define the $\wedge$ and $\setminus$ operator where the first argument is a matrix and the second is a row vector. The behaviour in this case is equivalent to replacing the vector with a matrix where every element in each row is equal to the corresponding element in the vector. Equivalently, these operations are also defined with column vectors.

We further define the *flattenRows* and *flattenColumns* operations, each taking a matrix and producing, respectively, a column and row vector. Each element of these vectors is 1 iff there is at least a 1 in the corresponding row or column, respectively.

The last operation that we need to define is *solveLocalMatchingProblem*, which given a local incidence matrix u returns a list of possible match matrices, called *match options*. More in detail, each of the returned options is a valid match, that satisfies the following properties:

(1) $m_{i,j} = 1 \implies u_{i,j} = 1$,
(2) Each option has at most one element at 1 for each row and column.

For example, applying *solveLocalMatchingProblem* to matrix

$$u = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \tag{7}$$

[1]A proof of concept implementation can be found at https://github.com/modelica-polimi/array-aware matching-poc

ALGORITHM 1: Array matching simplify algorithm.

```
Function simplify
    Input: $\underline{G} = (\underline{N} = \underline{V} \cup \underline{E}, \underline{D})$;
    Output: $\underline{G}$;
    $L \leftarrow \emptyset$;
    foreach $\underline{n} \in \underline{N}$ do
        if $\deg(\underline{n}) = 1$ then $L \leftarrow L \cup \{\underline{n}\}$;
    while $L \neq \emptyset$ do
        $\underline{n}_1 \leftarrow$ getElementIn($L$);
        $L \leftarrow L \setminus \{\underline{n}_1\}$;
        $\underline{n}_2 \leftarrow$ unmatched node reached from $\underline{n}_1$;
        $matchOptions \leftarrow$ solveLocalMatchingProblem($u_{12}$);
        if $|matchOptions| = 1$ then
            $alreadyMatched1 \leftarrow$ vector of size $|\underline{n}_1|$ where $alreadyMatched1_j = 1$ iff $\underline{n}_{1,j}$ is matched;
            $alreadyMatched2 \leftarrow$ vector of size $|\underline{n}_2|$ where $alreadyMatched2_j = 1$ iff $\underline{n}_{2,j}$ is matched;
            $m_{12} \leftarrow$ getElementIn($matchOptions$) $\setminus$ $alreadyMatched1 \setminus alreadyMatched2$;
            if all components of $\underline{n}_2$ are matched then
                $L \leftarrow L \setminus \{\underline{n}_2\}$;
                foreach $\underline{n}$ in $unmatchedReachedNodes(\underline{n}_2)$ do
                    if unmatchedDegree($\underline{n}$) = 1 then $L \leftarrow L \cup \{\underline{n}\}$;
            else
                if unmatchedDegree($\underline{n}_2$) = 1 then $L \leftarrow L \cup \{\underline{n}_2\}$;
```

returns the set of possible match matrices

$$\left\{ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right\} \tag{8}$$

where, depending on the *solveLocalMatchingProblem* implementation but without any effect on the proposed algorithms, the last match may or may not be returned as it can be considered part of the first (larger) one. As an additional constraint, *solveLocalMatchingProblem* shall try to return the largest possible match, that is, the one that matches the largest number of equation/variable pairs. This is not meant as a strong requirement – in other words, it is acceptable to provide an implementation of *solveLocalMatchingProblem* that does not return the largest possible match options in all cases. However, the larger the single options provided by this primitive, the higher quality the final matching will be.

7.2 Simplification Algorithm

The simplification step is dedicated to performing all the obligatory matches. Its importance in the handling of real-world problems becomes evident when considering that, as explained in [23, Chapter 7, and more specifically Section 7.2], when matching differential equations, the *state variables* of the system are considered to be known and need not be matched. Thus, the variables to be matched are either non-state variables, or derivatives of state variables. In modelling the evolution of physical systems over time, it is very common to write equations in the form

$$\dot{x}_i = \mathrm{f}(x, u)$$

where x and u are respectively the sets of state variables and inputs. These equations introduce equation nodes in the bipartite matching graph with a single arc connecting them to the corresponding derivative, and that thus can only be matched with one variable. For example, this applies to all the three arcs in Figure 1. The commonplace presence of obligatory matching options motivates the introduction of an efficient simplification step.

The proposed simplification step is shown in Algorithm 1. It takes as input the array graph $\underline{G}$, and operates as follows. First, it constructs a set L from every node in the graph with only one arc, regardless of it being an equation or a variable.

Then, for every node $\underline{n}_1$ in the set, it uses the *solveLocalMatchingProblem* procedure to attempt to match the only arc adjacent to that (array) node.

In the case of multiple matching options, the simplification step skips the considered node, because at this stage any matching choice would be arbitrary, and may affect the feasibility of the array-aware matching problem. Conversely, if only one option is found that that fully matches $\underline{n}_1$, that option is included in the solution. Attention then shifts to $\underline{n}_2$, the only node reached by $\underline{n}_1$.

Since we have fully matched one node and matched some variables from $\underline{n}_2$, the simplify procedure now checks whether also $\underline{n}_2$ is fully matched. If it is, $\underline{n}_2$ is removed from the set in case it was there, and since having fully matched $\underline{n}_2$ may result in neighboring nodes with only one unmatched arc, all such nodes are added to the set. Even if $\underline{n}_2$ is not fully matched, it too may end up having only one unmatched arc (if it previously had two), and in this case $\underline{n}_2$ is added to the set. The simplify algorithm thus recursively eliminates all nodes with a constrained match, leaving only irreducible connected components, as well as nodes where arcs have multiple matching options. After the simplify step all matches found are removed from the graph, and only the remaining part of the graph is passed to the subsequent matching algorithm.

7.2.1 Optimal Matching in Polynomial Time. It is evident from the formulation of the simplification step that its algorithm executes in polynomial time with respect to the number of nodes and arcs in the array graph and, with a suitable data structure to represent incidence matrices, in O(1) time with respect to the size of the arrays. We also notice that there is a class of array graphs that can be completely matched by application of the simplification algorithm alone. The model of a wire previously shown in Listing 1 is an example of that. By construction of the simplification algorithm, such graphs must have a single solution to the array-aware matching problem, and that solution is therefore optimal. It follows that such models can be optimally matched in polynomial time with respect to the number of nodes and arcs in the array graph, and in O(1) time with respect to the size of the arrays.

Finding other classes of graphs which can be optimally matched in polynomial time is an open research issue.

7.3 Matching Algorithm

Now we illustrate the complete array-aware matching algorithm. This algorithm must be able to match any valid array graph, and additionally it must attempt to approximate the optimal matching as much as possible. The algorithm we present is based on the well-known Hopcroft-Karp one for bipartite graph matching, with adaptations to support array graphs. Its main procedure is shown in Algorithm 2.

As we will see in the following, procedure *augmentingPaths* computes a list of non-intersecting *augmenting paths* in the graph to be applied later by the *applyPath* procedure. When no augmenting paths are found, the matching is considered complete.

Each augmenting path p, when applied, adds or removes matching from the graph. It consists of a list of tuples called *steps* $p_i = (\underline{n}, \underline{d}_{i,j}, m'_{i,j})$ where $\underline{n} \in \underline{N}$ is the starting node of the step, $\underline{d}_{i,j}$

ALGORITHM 2: Main procedure of the matching algorithm.

```
Function matching
    Input: $\underline{G} = (\underline{N} = \underline{V} \cup \underline{E}, \underline{D})$;
    Output: $\underline{G}$;
    P = augmentingPaths($\underline{G}$);
    while $P \neq \emptyset$ do
        foreach $p \in P$ do
            applyPath($G, p$)
        $P$ = augmentingPaths($G$);
```

ALGORITHM 3: Procedure for adding an augmenting path to the matching.

```
Function applyPath
    Input: $\underline{G}, p$;
    Output: $\underline{G}$;
    foreach $(\underline{n}, \underline{d}_{i,j}, m'_{i,j}) \in p$ do
        if $\underline{n}$ is the equation $\underline{e}_i$ then
            $m_{i,j} \leftarrow m_{i,j} \vee m'_{i,j}$
        else if $\underline{n}$ is the variable $\underline{v}_j$ then
            $m_{i,j} \leftarrow m_{i,j} \setminus m'_{i,j}$
```

is the arc being traversed, and $m'_{i,j}$ is the incidence matrix that specifies which elements of the local matching incidence matrix $m_{i,j}$ are modified by the step. In a similar way to the Hopcroft-Karp algorithm, steps starting from an equation node ($\underline{n} \in \underline{E}$) add non-zero matrix entries to the matching, and steps starting from a variable node ($\underline{n} \in \underline{V}$) remove entries from the matching. In other words, in a step starting from an equation node, the elements set to 1 in $m'_{i,j}$ are added to the matching matrix $m_{i,j}$. On the contrary, in a step starting from a variable node, the elements set to 1 in $m'_{i,j}$ are *removed* from $m_{i,j}$.

The computation of the augmenting paths is performed through a breadth-first-search in the residual graph. In contrast to the conventional scalar matching process performed by the Hopcroft-Karp algorithm, when computing the augmenting path we also need to keep track of the matching matrices that express the set of equivalent scalar variables and equations that are being matched or un-matched.

The breadth-first-search procedure must first be seeded with an initial frontier F by collecting the list of array equation nodes with at least one free scalar equation. This initial frontier is then passed to the *bfs* procedure, which computes both the initial list of augmenting path candidates L and the forest of search trees G_{BFS} traversed during the search. In G_{BFS}, A is the list of nodes in the forest, and B is the list of arcs. Each node $a \in A$ is a tuple (n, v) where n is a node in the matching graph, and v is a binary vector with size and dimensionality equivalent to the one of n specifying which scalar equations are being traversed in the path. Similarly, each arc $b \in B$ is a triple (a, m, a') where a is the parent node, a' the child node, and m is the local incidence matrix that describes the equivalent traversed path in the homomorphic scalar graph. The augmenting path candidates $l \in L$ are actually just leaves of the G_{BFS} forest.

In contrast to the basic Hopcroft-Karp matching algorithm, but similarly to the Ford-Fulkerson flow maximization algorithm [31], the initial "flow" in the first steps of the path can be different

ALGORITHM 4: Procedure for computing an augmenting path from a partial state of the matching.

```
Function augmentingPaths
    Input: $\underline{G}$;
    Output: $P$;

    (Calculation of the initial frontier)
    $F \leftarrow \{\}$;
    foreach $\underline{e}_i$ do
        $f \leftarrow$ vector of size $|\underline{e}_i|$ where $f_j = 1$ iff $\underline{e}_{i,j}$ is not currently matched;
        if $f \neq \emptyset$ then
            $F \leftarrow \text{append}(F, \{(\underline{e}_i, f)\})$;

    (Calculation of the augmenting paths with breadth-first-search)
    $G_{BFS} = (A, B), L \leftarrow \text{bfs}(G, F)$;

    (Heuristic sort of the augmenting paths)
    $L \leftarrow \text{heuristicSort}(L)$

    (Restriction of the flow of each augmenting path and removal of overlapping paths)
    $P \leftarrow \emptyset$;
    foreach $l_0 = (\underline{n}, s_0) \in L$ do
        $p \leftarrow \{\},\ l \leftarrow l_0,\ s \leftarrow s_0$;
        while $\exists (l', m, l) \in B$ do
            $l \leftarrow l',\ m' \leftarrow s \wedge m$;
            if $\underline{n}', \underline{n}$ are equations and variable $\underline{e}_i, \underline{v}_j$ then
                $s \leftarrow \text{flattenRows}(m')$;
                $p \leftarrow \text{append}(p, \{(\underline{e}_i, \underline{d}_{i,j}, m')\})$
            else if $\underline{n}', \underline{n}$ are variable and equation $\underline{v}_j, \underline{e}_i$ then
                $s \leftarrow \text{flattenColumns}(m')$;
                $p \leftarrow \text{append}(p, \{(\underline{v}_j, \underline{d}_{i,j}, m')\})$

        (Scalar arc intersection test between augmenting paths)
        if $\nexists \underline{e}_i \in \underline{E}, p' \in P : (\underline{e}_i, \underline{d}_{i,j}, m) \in p, (\underline{e}_i, \underline{d}_{i,j}, m') \in p', flattenRows(m) \wedge flattenRows(m') \neq \emptyset$
        $\wedge \nexists \underline{v}_j \in \underline{V}, p' \in P : (\underline{v}_j, \underline{d}_{i,j}, m) \in p, (\underline{v}_j, \underline{d}_{i,j}, m') \in p', flattenColumns(m) \wedge flattenColumns(m') \neq \emptyset$
        then
            $P \leftarrow \text{append}(P, \{p\})$;
```

from the final "flow" at the last step. In this context the "flow" of a given step is simply the number of scalar equations affected. To perform this operation, each path is traversed backwards – from the end to the beginning – and it is modified such that the following invariant is respected:

$$\begin{aligned} &\text{flattenRows}(m) = \text{flattenRows}(m') && \forall\ (a, m, a'), (a', m', a'') \in A : a' \text{ equation node} \\ &\text{flattenColumns}(m) = \text{flattenColumns}(m') && \forall\ (a, m, a'), (a', m', a'') \in A : a' \text{ variable node} \end{aligned}$$

During this process, we build the augmenting path p from the nodes, arcs, and matching matrices traversed.

Additionally, the breadth-first-search does not immediately return a set of paths that respect the *non-intersection condition* already present in the Hopcroft-Karp algorithm. An augmenting path intersects another if the two paths, in at least one point, traverse the same node with intersecting matching matrices. In order to guarantee this property, each augmenting path is tested against the others. If two paths are intersecting, one of the two is discarded.

Discarding intersecting paths has the effect of eliminating multiple candidates which are in mutual exclusion between each other. The specific candidates being discarded at each step influence the solution Ω, and thus how close the solution is to the optimum. Additionally, they impact the number of steps required by the matching algorithm. In our current implementation, such selection depends on the ordering of L. Our heuristic (implemented in the *heuristicSort* procedure) sorts L based on the number of ones in the matching matrices of the path (paths with more ones are prioritized). Other heuristics could be devised to improve the solution Ω and the number of steps required to complete the matching, and this could be an interesting direction for future work.

ALGORITHM 5: Breadth-first-search of augmenting paths in the matching graph.

```
Function bfs
    Input: $\underline{G}, F$;
    Output: $G_{BFS} = (A, B), L$;
    $F' \leftarrow \{\}$;
    $L \leftarrow \{\}$;
    while $F \neq \{\} \wedge P = \{\}$ do
        foreach $a = (\underline{n}, f) \in F$ do
            foreach $\underline{d}_{i,j}$ adjacent to $\underline{n}$ do
                if $\underline{n}$ is the equation $\underline{e}_i$ then move from equation to variable
                    $S \leftarrow$ solveLocalMatchingProblem($(u_{i,j} \setminus m_{i,j}) \wedge f$);
                    foreach $s \in S$ do
                        $t \leftarrow$ vertical vector of size $|\underline{v}_j|$ where $f_k = 1$ iff $\underline{v}_{j,k}$ is not currently matched;
                        $m \leftarrow s \wedge t$;
                        if $m \neq \emptyset$ then
                            $a' = (\underline{e}_i, \text{flattenRows}(m))$;
                            $A \leftarrow A \cup \{a'\},\ B \leftarrow B \cup \{(a, m, a')\},\ L \leftarrow \text{append}(L, \{a'\})$;
                        else
                            $a' = (\underline{e}_i, \text{flattenRows}(s))$;
                            $A \leftarrow A \cup \{a'\},\ B \leftarrow B \cup \{(a, s, a')\},\ F' \leftarrow \text{append}(F', \{a'\})$;
                else if $\underline{n}$ is the variable $\underline{v}_j$ then move from variable to equation
                    $S \leftarrow$ solveLocalMatchingProblem($m_{i,j} \wedge f$);
                    foreach $s \in S$ do
                        $a' = (\underline{v}_j, \text{flattenColumns}(s))$;
                        $A \leftarrow A \cup \{a'\},\ B \leftarrow B \cup \{(a, s, a')\},\ F' \leftarrow \text{append}(F', \{a'\})$;
        $F \leftarrow F',\ F' \leftarrow \{\}$;
```

The breadth-first-search procedure is shown in Algorithm 3. It operates in the conventional way, with three additional constraints:

(1) Each move in the search is associated with a *vector of tangent elements* f to the destination node $\underline{n}$ of the move. f is a vertical vector if $\underline{n}$ is an equation, otherwise f is an horizontal vector.

(2) Each move in the search must be associated with an incidence matrix s called *path matrix* that satisfies the same conditions imposed on matching matrices $m_{i,j}$ (see Section 5.1, conditions (3–6)).
(3) The incidence matrix s only affects the scalar variables or equations specified by f.

These constraints ensure that each path traversed during the search corresponds to a set of one or more equivalent paths in the scalar graph. As a result, multiple different moves can start at the same moment from the same node and through the same arc, but with different path matrices. The computation of the set of possible moves at each iteration is performed by enumerating the adjacent edges to the current node, and then by using *solveLocalMatchingProblem* to compute the set of distinct valid ways to traverse that edge. When a path reaches a variable node, and the path matrix contains at least one non-zero column not corresponding to any matched scalar variable, then it allows to augment the matching and the search is stopped. Since the search process is stopped for the entire frontier, all paths returned by the *bfs* procedure have the same length in terms of number of steps.

7.4 Data Representation to Achieve O(1) Complexity

To guarantee constant-time scaling with the size of array variables and equations, every operation performed on incidence matrices described in Section 7.1 can be completed in constant time with respect to the size of the matrices themselves.

However, it can be easily recognised that in reality this is not possible without at least setting an upper bound on the size of the matrices, and setting such a bound would limit the applicability of our approach to large equation systems (which is precisely our goal).

As a consequence, while multidimensional incidence matrices and vectors proved useful as a conceptual tool for explaining our methodology, as a data structure they are not suitable *as is* for implementation.

In order to achieve O(1) scaling we thus introduce two data structures to replace multidimensional incidence matrices and vectors, named ***Multidimensional Compressed Index Set*** **(MCIS)** and ***Multidimensional Compressed Index Map*** **(MCIM)**. The first data structure, the MCIS, replaces multidimensional vectors, and the second (MCIM) replaces multidimensional incidence matrices.

These two data structures are able to represent the entire range of possible vectors and matrices that can appear in a matching problem, but the operations on them are not constant-time in general. However, the operations are largely constant-time wherever the arrays in a model come from the spatial discretisation of partial-derivatives differential equations, which cover virtually the totality of the modelling cases of engineering interest.

Let us consider the set of multidimensional indices corresponding to the 1-elements of a multidimensional vector. Multidimensional Compressed Index Sets are data structures representing sets of multidimensional indices as lists of *multidimensional intervals* or *ranges*.

A *multidimensional range* defined over field $\mathbb{K} = \mathbb{N}^n$ is a list of tuples $\{(a_1, b_1), (a_2, b_2), \dots (a_n, b_n)\}$, one tuple for each dimension in $\mathbb{K}$. It represents the set

$$\{a_1, a_1 + 1, a_1 + 2, \dots b_1\} \times \{a_2, a_2 + 1, a_2 + 2, \dots b_2\} \times \cdots$$

where $\times$ indicates the Cartesian product. For example, the multidimensional range $\{(1, 3), (2, 4)\}$ represents the following set of indices:

$$\{(1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)\}$$

In other words, a multidimensional range is defined as a hyperrectangle over field $\mathbb{K}$ represented with the coordinates of its vertices.

An MCIS is a list of multidimensional ranges, and it represents the union of the hyperrectangles represented in turn by each range in the list. Additionally, the ranges do not intersect. A single multidimensional range of volume greater than 1 can also be represented as multiple adjacent ranges, and ranges may appear in any order. As a result, the same index set can be represented with MCISes in multiple ways.

Let us now consider the set of multidimensional indices corresponding to the 1-elements of a multidimensional incidence matrix. Multidimensional Compressed Index Maps are data structures representing this set of multidimensional indices, again as lists of intervals.

Each index in the set can be split in two parts: the sub-index corresponding to the first set of dimensions k, and the sub-index corresponding to the second set of dimensions j. For brevity we will represent each multidimensional index in the set as (k, j). Now, to obtain an MCIM from a set of indices $A = \{(k_1, j_1), (k_2, j_2), \ldots (k_n, j_n)\}$, first we split A in sub-sets $A_1, A_2, \ldots A_m$ where each index (k_i, j_i) satisfies the following identity:

$$\begin{aligned} & k_0 = \min_{1<i\leq n} k_i \\ & \delta = j_0 - k_0 \\ & j_i = k_i + \delta \qquad \forall i : 1 < i \leq n \end{aligned} \tag{9}$$

In other words, the j sub-indices in each subset A_i must be such that they can be obtained just from the corresponding k sub-indices and the δ_i, computed as described in Equation (9). δ_i must be constant for all elements of each subset, but may be different between one subset and another. To make another comparison for the sake of explanation, each subset A_i represents a diagonal of an incidence matrix.

At this point, for each sub-set $A_i = \{(k_{i,1}, j_{i,1}), \ldots (k_{i,n}, j_{i,n})\}$ consider the set $K_i = \{k_{i,0}, \ldots k_{i,n}\}$. We call an *MCIM element* the tuple $\Sigma_i = (K_i, \delta_i)$, where K_i is represented as a Multidimensional Compressed Index Set. Finally, an MCIM is the set of MCIM elements $\{\Sigma_1, \Sigma_2, \ldots \Sigma_m\}$, each of which corresponds to one of the sub-sets A_i.

The development of algorithms implementing the operations on the data structures we have described is largely an engineering problem, and we do not wish to delve into it. Depending on the specific algorithms being chosen, the complexity of most operations can range from an upper bound of $O(n^2)$, to $O(n)$ in the single-dimensional case if the list of indices in each MCIS is kept ordered. Better computational costs can be achieved by adopting well-known interval-tree representations [60], which allow operations on MCISes to reach $O(\log n)$ complexity in some specific cases.

However, independently from the implementation we choose for operations on MCISes, we can straightforwardly state that any operation on sets containing a single range can be implemented in constant time. The same holds for MCIMs with a single element Σ_1 containing a MCIS K_1 with a single range. This will be the case for any model where the matching heuristic algorithm manages to preserve all array equations and variables. Therefore, we expect that in such cases – which are typical in physical models [7] – the compilation process will be performed in constant time with respect to the size of array equations and variables.

The only operation that can be performed in constant $O(1)$ time on any arbitrary MCIM is *solveLocalMatchingProblem*. In fact, it is trivial to prove that each MCIM element Σ_i represents a matrix that satisfies the properties outlined in Section 7.1 for valid matrices returned by *solveLocalMatchingProblem*. Therefore, *solveLocalMatchingProblem* can be elided from the simplify and matching algorithms, and the iteration on the matching options can be replaced with the iteration of each Σ_i in the input MCIM.

8 CONCLUSIONS AND FUTURE WORK

We discussed the problem of translating declarative EB models into imperative code, concentrating on the crucial step of equation/variable matching. Relating our work to the research *scenario*, we evidenced two issues to address. First, currently established approaches to EB-to-imperative translation handle array variables and equation looping constructs in an extremely inefficient manner. Second, array-aware proposals in the literature aim for a fast translation, but not for obtaining an efficient imperative code.

We showed that to pursue both the objectives above, one needs to translate an EB model in such a way to maximise the preservation of looping constructs in the imperative code, and we introduced a metric to measure the said preservation. This led us to define the concept of *optimal* array-aware matching, as the one that maximises that metric.

As our main methodological contribution, we proved that the problem of computing an optimal matching in the sense just stated is NP-complete. Motivated by this completeness, we proposed an algorithm to compute an array-aware matching in polynomial time, whose *rationale* is to stop at the first solution found, privileging however the preservation of looping constructs when choices need to be taken.

The ideas we presented are currently being put to work within the implementation of an experimental Modelica compiler [7, 8]. Indeed, based also on the effort that such a realisation entails, we do hope that in the next years the developments we described will be adopted in the EB modelling community at large. In fact, the advantages of an array-aware EB-to-imperative translation are essential for addressing large-scale models in an industrial context.

In the future, we also plan to extend our approach to the rest of the translation pipeline, addressing other problems such as equation scheduling and SCC resolution. Additionally, a more precise characterization of the set of array graphs that can be optimally matched in polynomial time is also of great interest, as well as better heuristics for the array-aware matching algorithm.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their helpful suggestions that allowed us to improve the paper presentation, in particular regarding the definition of array equation and its impact on the optimality metric, as well as the opportunity to perform optimal matching in polynomial time for certain classes of models.

REFERENCES

[1] [n.d.]. Dymola home page. https://www.3ds.com/products-services/catia/products/dymola/
[2] [n.d.]. JModelica home page. https://jmodelica.org/
[3] [n.d.]. ModiaBase home page. https://github.com/ModiaSim/ModiaBase.jl
[4] [n.d.]. OpenModelica home page. https://www.openmodelica.org/
[5] [n.d.]. SimulationX home page. https://www.mathworks.com/products/connections/product_detail/simulationx.html
[6] Paul Serban Agachi, Zoltán K. Nagy, Mircea Vasile Cristea, and Árpád Imre-Lucaci. 2007. *Model Based Control: Case Studies in Process Engineering*. John Wiley & Sons, Hoboken, NJ, USA.
[7] Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. 2019. Towards a high-performance Modelica compiler. In *Proc. 13th International Modelica Conference*. Regensburg, Germany, 313–320. https://doi.org/10.3384/ecp19157313
[8] Giovanni Agosta, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. 2019. Towards a benchmark suite for high-performance Modelica compilers (work-in-progress paper). In *Proc. 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Berlin, Germany. https://doi.org/10.1145/3365984.3365988
[9] Abderrahim Ait-Alla, Markus Kreutz, Daniel Rippel, Micheel Lütjen, and Michael Freitag. 2019. Simulation-based analysis of the interaction of a physical and a digital twin in a cyber-physical production system. *IFAC-PapersOnLine* 52, 13 (2019), 1331–1336.
[10] Panagiotis Aivaliotis, Konstantinos Georgoulias, Zoi Arkouli, and Sotiris Makris. 2019. Methodology for enabling digital twin using advanced physics-based modelling in predictive maintenance. *Procedia CIRP* 81 (2019), 417–422.
[11] Mats Andersson. 1989. An object-oriented language for model representation. In *Proc. 2nd IEEE Control Systems Society Workshop on Computer-Aided Control System Design*. Tampa, FL, USA, 8–15.

[12] Mariano Asteasuain, Stella Maris Tonelli, Adriana Brandolin, and José Alberto Bandoni. 2001. Dynamic simulation and optimisation of tubular polymerisation reactors in gPROMS. *Computers & Chemical Engineering* 25, 4–6 (2001), 509–515.

[13] Karl Johan Åström and Wolfgang Kreutzer. 1986. System representations. In *Proc. 3rd IEEE Control Systems Society Symposium on Computer-Aided Control System Design*. Arlington, VA, USA, 8–15.

[14] Emanuele Baldino. 2018. *Structural Pitfalls of State-of-the-art Modelica Compilers: An Explorative Analysis*. MSc Thesis in Computer Science Engineering. Politecnico di Milano. http://hdl.handle.net/10589/144734

[15] Paul I. Barton and Constantinos C. Pantelides. 1994. Modeling of combined discrete/continuous processes. *AIChE Journal* 40, 6 (1994), 966–979.

[16] Jérémy Buisson and Nicolas Belloir. 2020. Digitalization in next generation C2: Research agenda from model-based engineering perspective. In *Proc. 15th IEEE International Conference of System of Systems Engineering*. Budapest, Hungary, 243–248.

[17] Peter Bunus and Karin Lunde. 2008. Supporting model-based diagnostics with equation-based object oriented languages. In *Proc. 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*. Paphos, Cyprus, 121–130.

[18] Francesco Casella. 2015. Simulation of large-scale models in Modelica: State of the art and future perspectives. In *Proc. 11th International Modelica Conference*. Versailles, France, 459–468.

[19] Francesco Casella, Andrea Giorgio Bartolini, and Alberto Leva. 2017. Equation-based object-oriented modelling and simulation of large-scale smart grids with Modelica. *IFAC-PapersOnLine* 50, 1 (2017), 5542–5547.

[20] Francesco Casella, Filippo Donida, and Marco Lovera. 2008. Beyond simulation: Computer aided control system design using equation-based object oriented modelling for the next decade. In *Proc. 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*. Paphos, Cyprus, 35–45.

[21] Francesco Casella and Alberto Leva. 2006. Modelling of thermo-hydraulic power generation processes using Modelica. *Mathematical and Computer Modelling of Dynamical Systems* 12, 1 (2006), 19–33.

[22] François E. Cellier. 1983. Simulation software: Today and tomorrow. In *Simulation in Engineering Sciences*, J. Burger and Y. Jarny (Eds.). Elsevier, Amsterdam, The Netherlands, 40–55.

[23] François E. Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer Science & Business Media, Berlin, Germany.

[24] Guo Qing Cheng, Bing Hai Zhou, and Ling Li. 2018. Integrated production, quality control and condition-based maintenance for imperfect production systems. *Reliability Engineering & System Safety* 175 (2018), 251–264.

[25] Chiara Cimino, Elisa Negri, and Luca Fumagalli. 2019. Review of digital twin applications in manufacturing. *Computers in Industry* 113, 11 (2019), 103130.

[26] Filippo Donida, Gianni Ferretti, Sergio Matteo Savaresi, and Mara Tanelli. 2008. Object-oriented modelling and simulation of a motorcycle. *Mathematical and Computer Modelling of Dynamical Systems* 14, 2 (2008), 79–100.

[27] Ulrich Drepper. 2007. *What Every Programmer Should Know About Memory*. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

[28] Hilding Elmqvist. 1979. DYMOLA – A structured model language for large continuous systems. In *Proc. Summer Computer Simulation Conference*. Toronto, Canada.

[29] Hilding Elmqvist and Sven Erik Mattsson. 1989. Simulator for dynamical systems using graphics and equations for modeling. *IEEE Control Systems Magazine* 9, 1 (1989), 53–58.

[30] John Fitzgerald, Peter Gorm Larsen, and Ken Pierce. 2019. Multi-modelling and co-simulation in the engineering of cyber-physical systems: Towards the digital twin. In *From Software Engineering to Formal Methods and Tools, and Back*, M. H. Ter Beek, A. Fantechi, and L. Semini (Eds.). Springer, Berlin, Germany, 40–55.

[31] Lester Randolph Ford and Delbert R. Fulkerson. 1956. Maximal flow through a network. *Canadian Journal of Mathematics* 8 (1956), 399–404.

[32] Avner Friedman. 1992. Sparse matrix methods for chemical process simulation. In *Mathematics in Industrial Problems. The IMA Volumes in Mathematics and its Applications*. Vol. 49. Springer, New York, NY, USA, 1–10.

[33] P. Fritzson, A. Pop, A. Asghar, B. Bachmann, W. Braun, R. Braun, L. Buffoni, F. Casella, R. Castro, and A. Danós. 2019. The OpenModelica integrated modeling, simulation and optimization environment. In *Proc. American Modelica Conference 2018*. Cambridge MA, USA, 206–219.

[34] Peter Fritzson, Adrian Pop, David Broman, and Peter Aronsson. 2009. Formal semantics based translator generation and tool development in practice. In *Proc. 2009 Australian Software Engineering Conference*. Gold Coast, QLD, Australia, 256–266.

[35] Alberto García, Luis Felipe Acebes, and Cesar de Prada. 2002. Modelling and simulation of batch processes: A case study. *IFAC Proceedings Volumes* 35, 1 (2002), 169–174.

[36] Michael R. Garey, David S. Johnson, and Larry Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoretical Computer Science* 1, 3 (1976), 237–267.

[37] Bin He, Long Liu, and Dong Zhang. 2021. Digital twin-driven remaining useful life prediction for gear performance degradation: A review. *Journal of Computing and Information Science in Engineering* 21, 3 (2021), 030801.

[38] Alan C. Hindmarsh, Peter N. Brown, Keith E. Grant, Steven L. Lee, Radu Serban, Dan E. Shumaker, and Carol S. Woodward. 2005. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Software* 31, 3 (2005), 363–396.

[39] Gerd Hirzinger, Johann Bals, Martin Otter, and Johannes Stelter. 2005. The DLR-KUKA success story: Robotics research improves industrial robots. *IEEE Robotics & Automation Magazine* 12, 3 (2005), 16–23.

[40] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231. https://doi.org/10.1137/0202019. arXiv:https://doi.org/10.1137/0202019

[41] Kenny Hunt and James Cremer. 1997. Refiner: A problem solving environment for ODE/DAE simulations. *ACM SIGSAM Bulletin* 31, 3 (1997), 42–43.

[42] Sukumar Jairam, Kusum Lata, Subir K. Roy, and Navakanta Bhat. 2008. Verification of a MEMS based adaptive cruise control system using simulation and semi-formal approaches. In *Proc. 15th IEEE International Conference on Electronics, Circuits and Systems*. Dubai, UAE, 910–913.

[43] Samir Khan, Michael Farnsworth, Richard McWilliam, and John Erkoyuncu. 2020. On the requirements of digital twin-driven autonomous maintenance. *Annual Reviews in Control* 50 (2020), 13–28.

[44] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *SIGPLAN Not.* 35, 5 (May 2000), 145–156. https://doi.org/10.1145/358438.349320

[45] Le-Wei Li. 2010. Integral-equation based fast solvers developed for microwave and RF applications. In *Proc. 40th European Microwave Conference*. Paris, France, 545–548.

[46] Kilian Link, Haiko Steuer, and Axel Butterlin. 2009. Deficiencies of Modelica and its simulation environments for large fluid systems. In *Proc. 7th International Modelica Conference*. Como, Italy, 341–344.

[47] Marco Macchi, Irene Roda, Elisa Negri, and Luca Fumagalli. 2018. Exploring the role of digital twin for asset lifecycle management. *IFAC-PapersOnLine* 51, 11 (2018), 790–795.

[48] Robert Malone, Brittany Friedland, John Herrold, and Daniel Fogarty. 2016. Insights from large scale model based systems engineering at Boeing. In *Proc. 26th INCOSE International Symposium*. Edinburgh, UK, 542–555.

[49] Juan José Marquez, Ascensión Zafra-Cabeza, and Carlos Bordons. 2018. Diagnosis and fault mitigation in a microgrid using model predictive control. In *Proc. 2018 International Conference on Smart Energy Systems and Technologies*. Sevilla, Spain, 1–6.

[50] Sven Erik Mattsson, Hilding Elmqvist, and Jan F. Broenink. 1997. Modelica: An international effort to design the next generation modelling language. *Journal A* 38, 3 (1997), 16–19.

[51] Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. 1998. Physical system modeling with Modelica. *Control Engineering Practice* 6, 4 (1998), 501–510.

[52] Ricardo Mejía-Gutiérrez and Ricardo Carvajal-Arango. 2017. Design verification through virtual prototyping techniques based on systems engineering. *Research in Engineering Design* 28, 4 (2017), 477–494.

[53] Dieter Moormann and Gertjan Looye. 2002. The Modelica flight dynamics library. In *Proc. 2nd International Modelica Conference*. Oberpfaffenhofen, Germany, 275–284.

[54] Jakob Mund, Safa Bougouffa, Iman Badr, and Birgit Vogel-Heuser. 2018. Towards verified continuous integration in the engineering of automated production systems. *At-Automatisierungstechnik* 66, 10 (2018), 784–794.

[55] Tom Oomen and Maarten Steinbuch. 2020. Model-based control for high-tech mechatronic systems. *Mechatronics and Robotics* (2020), 51–80.

[56] Martin Otter and Hilding Elmqvist. 2017. Transformation of differential algebraic array equations to index one form. In *Proceedings of the 12th International Modelica Conference*. Linköping University Electronic Press, 565–579. https://doi.org/10.3384/ecp17132565

[57] Jorge Ricardo Paloschi. 1996. Using sparse bounded homotopies in steady state simulation packages. *Computers & Chemical Engineering* 20 (1996), S285–S290.

[58] Constantinos C. Pantelides. 1988. The consistent initialization of differential-algebraic systems. *SIAM J. Sci. Statist. Comput.* 9, 2 (1988), 213–231.

[59] Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, and Rüdiger Franke. 2019. A new OpenModelica compiler high performance frontend. In *Proceedings of the 13th International Modelica Conference*. Linköping University Electronic Press, 689–698. https://doi.org/10.3384/ecp19157689

[60] F. P. Preparata and M. I. Shamos. 1985. *Computational Geometry: An Introduction*. Springer-Verlag New York. https://doi.org/10.1007/978-1-4612-1098-6

[61] François-Henry Rouet, Cleve Ashcraft, Jef Dawson, Roger Grimes, Erman Guleryuz, Seid Koric, Robert F. Lucas, James S. Ong, Todd A. Simons, and Ting-Ting Zhu. 2020. Scalability challenges of an industrial implicit finite element code. In *Proc. 34th IEEE International Parallel and Distributed Processing Symposium*. New Orleans, LA, USA, 505–514.

[62] Per Sahlin, Lars Eriksson, Pavel Grozman, Hans Johnsson, Alexander Shapovalov, and Mika Vuolle. 2003. Will equation-based building simulation make it?-experiences from the introduction of IDA Indoor Climate And Energy. In *Proc. 8th International IBPSA Conference*. Eindhoven, The Netherlands, 1147–1154.

[63] Anders Sandholm, Peter Bunus, and Peter Fritzson. 2006. A numeric library for use in Modelica simulations with LAPACK, SuperLU, Interpolation, and MatrixIO. In *Proc. 5th International Modelica Conference*. Vienna, Austria, 285–292.

[64] Michael Schluse, Maec Priggemeyer, Linus Atorf, and Jürgen Rossmann. 2018. Experimentable digital twins—Streamlining simulation-based systems engineering for Industry 4.0. *IEEE Transactions on Industrial Informatics* 14, 4 (2018), 1722–1731.

[65] Joseph Schuchart, Volker Waurich, Martin Flehmig, Marcus Walther, Wolfgang E. Nagel, and Ines Gubsch. 2015. Exploiting repeated structures and vectorization in Modelica. In *Proceedings of the 11th International Modelica Conference*. Linköping University Electronic Press, 265–272. https://doi.org/10.3384/ecp15118265

[66] Gerald Schweiger, Henrik Nilsson, Josef Schoeggl, Wolfgang Birk, and Alfred Posch. 2020. Modeling and simulation of large-scale systems: A systematic comparison of modeling paradigms. *Appl. Math. Comput.* 365 (2020), 124713.

[67] Martin Sjölund. 2015. *Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models*. Vol. 1664. Linköping University Electronic Press, Linköping, Sweden.

[68] Edward F. Sowell and Philip Haves. 2001. Efficient solution strategies for building energy system simulation. *Energy and Buildings* 33, 4 (2001), 309–317.

[69] Mark A. Stadtherr and E. Stephen Wood. 1984. Sparse matrix methods for equation-based chemical process flowsheeting – I: Reordering phase. *Computers & Chemical Engineering* 8, 1 (1984), 9–18.

[70] Mark A. Stadtherr and E. Stephen Wood. 1984. Sparse matrix methods for equation-based chemical process flowshweeting – II: Numerical phase. *Computers & Chemical Engineering* 8, 1 (1984), 19–33.

[71] Yoshihiko Susuki and Takashi Hikihara. 2002. An analysis with differential-algebraic equation in electric power system with DC transmission. In *Proc. 2002 IEEE/PES Transmission and Distribution Conference and Exhibition*, Vol. 3. Yokohama, Japan, 1933–1936.

[72] Fei Tao, Meng Zhang, Yushan Liu, and Andrew Y. C. Nee. 2018. Digital twin driven prognostics and health management for complex equipment. *CIRP Annals* 67, 1 (2018), 169–172.

[73] Robert E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1 (1972), 146–160.

[74] Tayfun E. Tezduyar, Shahrouz K. Aliabadi, Michael J. Behr, Arthur G. Johnson, Vinay J. Kalro, and Matthew Litke. 1996. High performance computing in flow simulations. *Computer Methods in Applied Mechanics and Engineering* 134 (1996), 351–373.

[75] Job Van Amerongen and Peter Breedveld. 2003. Modelling of physical systems for the design and control of mechatronic systems. *Annual Reviews in Control* 27, 1 (2003), 87–117.

[76] Dirk A. van Beek, Wan J. Fokkink, Dennis Hendriks, Albert T. Hofkamp, Jasen Markovski, Joanna M. van de Mortel-Fronczak, and Michel A. Reniers. 2014. CIF 3: Model-based engineering of supervisory controllers. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Grenoble, France, 575–580.

[77] Jacques Verriet, Lennart Buit, Richard Doornbos, Bas Huijbrechts, Kristina Ševo, Jack Sleuters, and Mark Verberkt. 2019. Virtual prototyping of large-scale IoT control systems using domain-specific languages. In *Proc. 7th International Conference on Model-Driven Engineering and Software Development*. Praha, Czech Republic, 229–239.

[78] Wenqing Wang and Olaf Kolditz. 2010. Sparse matrix and solver objects for parallel finite element simulation of multi-field problems. In *High Performance Computing and Applications*, W. Zhang and Z. Chen (Eds.). Springer, Berlin, Germany, 418–425.

[79] Michael Wetter and Christoph Haugstetter. 2006. Modelica versus TRNSYS – A comparison between an equation-based and a procedural modeling language for building energy simulation. In *Proc. 2nd SimBuild Conference*. Cambridge, MA, USA.

[80] Pablo Zimmermann, Joaquín Fernández, and Ernesto Kofman. 2019. Set-based graph methods for fast equation sorting in large DAE systems. In *Proc. 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools*. Berlin, Germany, 45–54.

Received 14 October 2021; revised 17 November 2022; accepted 17 July 2023