Efficient Memory Management for Modelica Simulations

Michele Scuttari
Politecnico di Milano, Italy

Nicola Camillucci
Politecnico di Milano, Italy

Daniele Cattaneo
Politecnico di Milano, Italy

Federico Terraneo
Politecnico di Milano, Italy

Giovanni Agosta
Politecnico di Milano, Italy

Abstract

The ever increasing usage of simulations in order to produce digital twins of physical systems led to the creation of specialized equation-based modeling languages such as Modelica. However, compilers of such languages often generate code that exploits the garbage collection memory management paradigm, which introduces significant runtime overhead. In this paper we explain how to improve the memory management approach of the automatically generated simulation code. This is achieved by addressing two different aspects. One regards the reduction of the heap memory usage, which is obtained by modifying functions whose resulting arrays could instead be allocated on the stack by the caller. The other aspect regards the possibility of avoiding garbage collection altogether by performing all memory lifetime tracking statically. We implement our approach in a prototype Modelica compiler, achieving an improvement of the memory management overhead of over 10 times compared to a garbage collected solution, and an improvement of 56 times compared to the production-grade compiler OpenModelica.

2012 ACM Subject Classification Software and its engineering → Compilers; Computing methodologies → Modeling and simulation

Keywords and phrases Modelica, modeling & simulation, memory management, garbage collection

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.7

1 Introduction

The explosion of Industry 4.0 has driven a renewed interest in the topic of modeling and simulation, due to a significant increase in complexity of systems that need to be simulated. At the same time, simulation is nowadays used for a range of vital taks in the lifecycle of industrial products, generally subsumed under the moniker of *digital twins* [16, 5, 15]. Digital twins promise the ability to perform a wide range of experiments, assessments, and predictions on real-world physical systems, such as cars, planes, buildings and power-distribution networks. The phenomena to be modeled in digital twins are natively expressed in terms of Differential and Algebraic Equations (DAE). Those equations need to be translated into an imperative simulation program performing numerical integration by means of well-known mathematical methods. The resulting program is then executed to obtain the evolution of the system during a period of time.

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 7; pp. 7:1–7:13

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Equation-based modeling languages are declarative languages that directly allow to input a system of equations, taking advantage of the always-increasing computing power by performing automated model translation. This approach relieves the modeler from the time-consuming and error-prone task of manually translating models of increasing complexity into the corresponding algorithmic solution code. One of the most popular modeling languages is Modelica [17]. Other than allowing to define DAE systems, it also encompasses the possibility to define functions in a way similar to imperative programming languages. This capability is useful in Cyber Physical Systems (CPS) to model the algorithmic part, such as controllers, as well as to express numerical correlation or tabular material properties – such as the fluid ones [8].

Just like in regular programming languages, Modelica functions can also receive and return arrays. More in detail, arrays in Modelica always have a fixed number of dimensions – that we will call *rank* – but the size of each of them can either be fixed or change during the execution of the simulation.

For what regards memory management, Modelica does not provide explicit access to pointers but rather consider arrays as objects without specifying how implementations should handle array copying. This is in accordance with the Modelica objective, that is to provide a high-level language to model complex systems.

The absence of explicit memory management is usually addressed by Modelica compilers by leveraging garbage collectors [20], which take care of periodically analysing memory reachability and deallocate the blocks detected as no longer in use. This technique has been widely tested throughout the years, but even if effective it is not optimal in terms of performance [19]. The overheads incurred from usage of garbage collection are particularly significant in the context of embedded systems. In particular, they hamper the usability of Modelica and similar languages for automatically producing system mock-ups as proposed by the eFMI open standard [14], which has been growing in importance in the last few years.

Our contribution consists in the introduction of two optimization passes that operate on the LLVM-IR produced by a prototype LLVM-based compiler, but can nonetheless be applied to other intermediate representations. The first one enables the allocation on the stack of all fixed-size arrays, independently from their usage as argument or result within a function. The second pass addresses the deallocation of the dynamically-sized – and thus heap-allocated – ones, by introducing the deallocation instructions in the right positions within the IR and thus avoiding the need for garbage collection.

The document is organized as follows. In section 2 we describe the semantics of the Modelica languages that are useful for this work and also briefly discuss the state of the art in Modelica compilers, with a focus on the memory management aspect. In section 3 we describe our code-generation strategy for memory management, in particular a transformation by which all the fixed-size arrays can be potentially allocated on the stack, and how to correctly place the deallocation instructions for heap-allocated arrays in order to remove the need for garbage-collection. Then, in section 4 we examine the correctness of our approach by using a prototype compiler based on LLVM [13]. We also make some comparisons with another industry-grade compiler and with a customized version of our compiler that uses the Boehm garbage collector. Finally, in section 5 we review the results and we discuss future directions for improvements of Modelica compilers.

2 Background

Declarative modeling languages include Modelica, gPROMS [6], Simscape [22] and Omola [4]. Of these, Modelica is one of the most popular, providing a separate language specification that is implemented by both open source and commercial simulation environments. In the first category we find OpenModelica [10], while in the second we find Dymola [9] and JModelica [3]. These languages allow the modelers to focus on the description of the systems and delegate the implementation details, such as memory management, to the model translator or compiler.

In general, a Modelica program consists of a set of DAE equations describing the evolution in time of the physical system to simulate. These equations involve a fixed set of variables, among which we can find the system *state variables*, that represent the current state of the system at a given moment in time. All these have the exact same lifetime as the simulation itself, thus obviating the need for memory management.

However, Modelica also allows to define functions as in common imperative programming languages. Functions receive some arguments and return others. The return values are computed by executing the imperative code in the body of the function. In accordance with the language specification, we will call the former *input values* and the latter *output values*. Within the body of a function it is not allowed to modify the input values, while the output ones can be rewritten as many times as needed. Inside the function it is also possible to define *protected* values living only within the function; their modifications obey to the same rules of output ones.

An example is shown in code listing 1. The *foo* function declares three input arrays, two scalar input values, one output array and one protected array. All the arrays of this example are characterized by a size that is potentially known only at runtime. In the body of the function, the first assignment at line 9 consists in the sum of three input arrays: the first sum between a and b creates a temporary array that is summed with c; the resulting array value is assigned to the internal array d. At line 10, the values within d are doubled and stored into e. Then, according to the *expand* value, the array e may be set with the values returned by the *bar* function call. The body of *bar* is not relevant and thus omitted, but the signature clearly shows how the dimension of the output array y is different from the size of e as computed up to this point. According to the language specification, all the assignments to d and e are legit, as they write into a protected and an output variable, and at the same time they also determine their size. Finally, a loop increments all elements in e by the input value v. From this simple example we notice several peculiarities that set Modelica aside from many other imperative languages. Specifically, the fact that an assignment to an array may determine or change its size at runtime, the immutability of input values and the possibility of creating complex expression with array operands. Moreover, the assignments of arrays in Modelica have the same semantics as an element-by-element copy – in other words, array types are not references.

While these peculiarities are important to understand the semantics of the language, our compiler operates on an LLVM intermediate representation (LLVM-IR) of the program that follows its own specific rules, which are closer to assembly languages. Modelica-specific semantics are enforced by previous stages within the compilation pipeline which generate this intermediate representation. In order to avoid any confusion, unless otherwise specified, in the following sections we will always refer to constructs implemented in, and with the semantics of, LLVM-IR.

Listing 1 Modelica function example.

```
1  function foo
2      input Real[:] a, b, c;
3      input Boolean expand;
4      input Real v;
5      output Real[:] e;
6  protected
7      Real[:] d;
8  algorithm
9      d := a + b + c;
10     e := d * 2;
11
12     if expand then
13         e := bar(d);
14     end if;
15
16     for i in 1:size(e,1) loop
17         e[i] := e[i] + v;
18     end for;
19 end foo;
20
21 function bar
22     input Real[:] x;
23     output Real[size(x,1) * 2] y;
24 end bar;
```

In contrast to most imperative programming languages, the Modelica language specification[1] does not specify any particular memory management paradigm, but just the expected lifetime of each variable and the value semantics. Therefore, complete freedom is left for the implementor to choose a memory management approach that satisfies the semantics of the language.

Even though functions can be defined, the semantics of Modelica are not enough for general-purpose programming. An extension of Modelica, called MetaModelica [21], has been devised to make the language powerful enough for programming applications. This language extension introduces some constructs – such as lists and exceptions – which do not belong to the standard Modelica specification but allow for the OpenModelica compiler to be self-hosting. MetaModelica shares many design aspects with functional languages [11] – as a result, memory management using garbage collection is a natural design choice. This decision also permeated into the simulation programs generated by OpenModelica, which make use of the Boehm garbage collector even for standard Modelica models not using any MetaModelica language construct. A recent study aims to introduce the LLVM infrastructure into the backend of OpenModelica in order to translate both MetaModelica and Modelica into LLVM-IR instead of C code, but the garbage-collected nature of the language implementation has been retained [23] in order to support MetaModelica.

[1] https://specification.modelica.org/maint/3.5/MLS.html

Table 1 Calling conventions used for Modelica functions within LLVM-IR after the execution of the discussed passes.

	Input	Output
Scalar	By value	By value
Fixed array	Pointer to memory allocated by caller	[Promoted to input]
Dynamic array	Pointer to memory (dynamically) allocated by caller	Pointer to memory (dynamically) allocated by callee

3 Proposed solution

In this section we analyze the two transformation passes that we implemented in our prototype Modelica compiler.

The initial intermediate representation given as input to the passes has the following characteristics: all output arrays are allocated on the heap and there are no deallocation instructions; the input arrays are either the ones of state variables (which are placed on the heap in order to live throughout the whole simulation), the ones returned by function calls (thus, heap-allocated) or protected ones (which are allocated on the stack if their size is fixed, or on the heap otherwise).

We will first focus on the output arrays and describe a transformation pass by which some of them may become stack-allocated. We will then examine the remaining dynamically sized arrays and determine how to correctly place deallocation instructions in order to avoid garbage collection.

3.1 Promotion of Output Arrays

Depending on how an array is used by a function, its memory allocation strategy may change. For example, in the C programming language, an array declared within a function is allocated on the call stack. Therefore, the array ceases to exist automatically as soon as the function terminates. Instead, if the array must outlive the function where it is created, it must be either dynamically allocated on the heap, or allocated on the stack beforehand by the caller of the function. These semantics of the C language map directly to LLVM-IR and machine code.

In the context of Modelica, arrays outliving the function where they are declared are always denoted as output parameters – input parameters are immutable. A compiler can avoid generating heap allocations for such arrays by creating the corresponding allocation on the stack before each call to the function. As a result, in the implementation, fixed size output arrays become mutable input arrays passed by reference, a construct that cannot be directly expressed in the Modelica language. This transformation is called *promotion of output arrays*.

In order to perform this transformation, we implement a pass which identifies the output arrays that can be potentially allocated on the stack and converts them into arguments to the function. The promotion may be applicable or not depending on multiple factors, such as the overall size of the array (which could lead to a stack overflow) or whether the function is a recursive one. For the sake of simplicity, we will limit our strategy to just consider the fixed or dynamic nature of the array, as visible in table 1.

Algorithm 1 Output-to-input promotion pass for functions.

```
Function promoteFunctionResults
    Input: f : Function
    L ← ∅
    foreach t_i ∈ resultTypes(f) do
        if canBePromoted(t_i) then
            newArgument ← addArgumentWithType(f, t_i)
            L ← L ∪ {i}
            allocation ← getAllocInstruction(f, i)
            replaceUsages(allocation, newArgument)
        end
    end
    removeResultsFromSignature(f, L)
    returnInstruction ← getReturnInstruction(f)
    results ← {}
    foreach v_i ∈ arguments(returnInstruction) do
        if i ∉ L then
            results ← append(results, v_i)
        end
    end
    setArguments(returnInstruction, results)
end
```

While performing this transformation, function call sites also need to be updated so that they reflect the updated function signatures.

The transformation pass is described by algorithms 1 and 2. The *promoteFunctionResults* function modifies each function signature and definition, while the *promoteCallsResults* function updates the calls to those functions. Some utility methods are used within the pseudo-code and their implementation depends on the characteristics of the intermediate representation being used. Some of them are self-explanatory, while the remaining ones are defined as follows:

addArgumentWithType(f, t). Append a new argument with type t to the signature of function f and returns the newly added argument.

alloca(t). Allocate on the stack a value with type t.

canBePromoted(t). Determine whether a value with type t can be placed on the stack.

getAllocInstruction(f, i). Get the allocation instruction that is used within the body of function f to create the result with index i. The index consists in the position of the result among the original ones.

removeResultsFromSignature(f, I). Remove the results with the positions given by set I from the signature of function f.

replaceUsages(op, V). Replace all the usages of the results of instruction *op* with the values of set V; the original instruction *op* is also eliminated.

At the end of the transformation pass all the fixed-size output arrays have become stack-allocated by the caller. As a consequence, the only fixed-size arrays that are left on the heap are the ones related to the state variables, which need to live throughout the whole simulation.

3.2 Heap Array Deallocation

As we have just seen, fixed-size arrays can be potentially always allocated on the stack. Dynamically-sized arrays, on the contrary, must be allocated on the heap because their size is known only at runtime. Static analysis may simplify some cases in which their size can be inferred at compile time to be fixed, but yet the rule holds for the most generic case.

Differently from arrays placed on the stack, heap-allocated arrays are not automatically released and thus explicit deallocations must take place. The pass we are going to describe aims to insert such deallocations in the correct positions, so that all the arrays are deallocated exactly once and only when they are not used anymore.

It must be noted that, since array expressions are allowed in Modelica, intermediate values of such expressions are also arrays. These arrays might be dynamically allocated if the size of the array operands is unknown, but after the initial allocation their size is fixed. On the contrary, dynamic arrays declared by the programmer may require a reallocation due to resizing at runtime. In order to implement this behaviour, the underlying buffer that stores its content must be replaceable during the execution of the simulation. For this reason, this last kind of arrays is represented by means of a pointer to pointer. When a reallocation happens the new pointer is stored in such data structure, overwriting the older pointer.

However, overwriting the previous address would lead to memory leaks, as no reference to the previously addressed memory would exist anymore. In order to avoid this issue, the first part of this transformation pass takes care of finding the store operations overwriting the address, and, right before each of them, place a deallocation instruction for the address that is going to be overwritten. The first run-time deallocation would indeed be illegal as no previous memory would have been reserved yet, but a simple check on the pointer validity is sufficient to avoid this failure.

For what regards the temporary dynamic arrays, we have already seen how their size is determined at runtime but yet will never change. For this reason, they are not referenced by a pointer to a pointer. However, the deallocation must take also aliases into consideration. An example of aliasing is subscription, which creates a reduced-rank view over the original array but without allocating further memory.

Algorithm 3 shows the procedure to be applied in order to retrieve the list of heap-allocated arrays and their aliases. The *arrayAndAliases* procedure takes the function to be analyzed and returns the sets L and A: the former contains the SSA values representing the heap-allocated arrays we need to handle; the latter consists in pairs of values mapping each alias to the aliased array. Moreover, an allocation is considered as an alias of itself. In case of nested sub-views, A maps to the view being aliased and not to the original array. Some utility functions have also been used within the algorithm, and reported here for clarity:

isAlias(v). check if the value v is an alias for some other value.

shouldBeDeallocated(v). check if value v is heap-allocated and does not belong to the set of arrays created by reallocations (which are already handled, as explained earlier).

The placement of the deallocation instructions takes place accordingly to function *placeDeallocations* of algorithm 4, which is applied to each function within the IR. The definition of the most important utility functions leveraged within the algorithm are the following:

createDeallocationAfter(v, op). Create the deallocation instruction for value v right after instruction *op*.

findCommonPostDominator(aliases). Find the block that post-dominates all the blocks in which the values contained in the set *aliases* are defined. This requires the capability to compute the dominance information regarding the blocks of the function; being this a well known dataflow analysis [12], we will not explore its implementation details.

Algorithm 2 Output-to-input promotion pass for function calls.

```
Function promoteCallsResults
    Input: call : Instruction
    args ← arguments(call)
    newArgs ← {}
    promoted ← ∅
    filteredResultTypes ← {}
    n ← numResults(call)
    foreach t_i ∈ resultTypes(call) do
        if canBePromoted(t_i) then
            newArgs ← append(newArgs, alloca(t_i))
            promoted ← promoted ∪ {i}
        else
            filteredResultTypes ← append(resultTypes, t_i)
        end
    end
    args ← append(args, newArgs)
    newCall ← createCall(callee(f), args, filteredResultTypes))
    results ← {}
    j ← 0
    k ← 0
    for i = 0, ... n do
        if i ∈ promoted then
            results ← append(results, newArgs[j])
            j ← j + 1
        else
            results ← append(results, result(newCall, k))
            k ← k + 1
        end
    end
    replaceUsages(call, results)
end
```

findLastUsageInBlock(v, b). Get the last operation within the block b that has the value v among its arguments.

isBefore(op1, op2). Check if the operation *op1* is placed before the operation *op2* within the IR; the two operations are assumed to belong to the same block.

4 Experimental Evaluation

In order to prove the correctness of our approach, we use a benchmark Modelica model describing a series of heat exchangers operating with methanol in the gaseous phase as the working fluid. This models makes use of functions to model the methanol properties. These functions have been written in three different forms, leading to three different models: the first one operates only with scalar values, the second uses arrays with fixed size, and the third covers the more generic case of dynamically-sized arrays.

All the tests have been performed on a Linux machine with the following hardware characteristics and software setup:

- **OS**: Ubuntu 20.04
- **CPU**: Intel Xeon CPU E5-2650 2.30GHz
- **RAM**: 72 GB DDR3 2133 MHz
- LLVM 13.0.0
- OpenModelica v1.19.0-dev.392+g2ca59e4f7e

Algorithm 3 Array and aliases discovery.

```
Function arraysAndAliases
    Input: f : Function
    Output: L : Set, A : Set
    L ← ∅
    A ← ∅
    foreach op ∈ operations(f) do
        v = result(op)
        if shouldBeDeallocated(v) then
            L ← L ∪ {v}
            A ← A ∪ {(v, v)}
        else if isAlias(v) then
            s ← aliasedValue(v)
            if ∃(a, b) ∈ A : b == s then
                A ← A ∪ {(s, v)}
            end
        end
    end
end
```

For what regards the simulation options, all the models have been simulated using the forward Euler method with a time step of 0.01s for a total amount of 1 000 000 steps.

4.1 LLVM-based prototype compiler

The LLVM-based prototype compiler was developed starting from an already existing one that was used to demonstrate the limitations of current solutions [1]. A profiling system was also introduced in order to keep track of the number of heap allocations and deallocations executed during the simulation, together with the time spent in doing such operations. This allowed to verify that the number of allocations is equal to the deallocations one, and thus ensuring that no memory leak or double deallocation happens. Valgrind [18] has also been leveraged to confirm this result, and it indeed showed the absence of definitely, indirectly or possibly lost references.

Furthermore, we also created a custom version of our compiler leveraging the Boehm garbage collector and we compared its performance with the original implementation. Table 2 shows the measurements for what regards the total execution time and the time spent during the heap memory management. The values have been computed on an average of 1 000 executions.

The version without garbage collection showed a speed-up of 6.5% for what regards the total execution time. The time spent in the heap management reported an improvement of a factor $\sim$ 13. One may argue that the $\sim$ 2 seconds difference of the total execution should perfectly reflect within the heap management. However, the latter does not take into consideration the overhead of the creation and destruction of the GC-related structures, which happen at the beginning and at the end of the simulation and thus are not captured by the profiling of the individual allocation instructions.

Finally, the Valgrind tool has again been used to measure the peak heap-allocated memory with and without garbage collection. No significant differences were observed, in both cases the measurement is approximately 446 KB.

Algorithm 4 Placement of deallocation instructions.

```
Function placeDeallocations
    Input: f : Function
    (L, A) ← arraysAndAliases(f)
    foreach array ∈ L do
        aliases ← ∅
        aliasQueue ← {array}
        while ¬ empty(aliasQueue) do
            current = popFront(aliasQueue)
            foreach (a, b) ∈ A : a == current do
                aliasQueue ← append(aliasQueue, b)
            end
        end
        block ← findCommonPostDominator(aliases)
        lastUsage = firstOp(block)
        foreach a ∈ aliases do
            u = findLastUsageInBlock(a, block)
            if isBefore(lastUsage, u) then
                lastUsage ← u
            end
        end
        createDeallocationAfter(array, lastUsage)
    end
end
```

Table 2 Execution times for the model with dynamically sized arrays, compiled with OpenModelica (OM) and the Prototype LLVM-based compiler.

	OpenModelica	**Prototype**	
Garbage collection	Yes	Yes	No
Total execution time [s]	74.19	27.36	25.58
Heap management time [s]	7.23	1.77	0.13
Heap management fraction [%]	9.75	6.47	0.51

4.2 Comparison with OpenModelica

Considering the third model – that is the most interesting one, with dynamically sized arrays – we compared the simulation generated by our prototype compiler with the one given by OpenModelica. As in the previous section, we measured the total execution time and the time spent in heap memory management on an average of 1 000 executions.

OpenModelica is known to be affected by some inefficencies in handling large-scale models [7, 1, 2]. In fact, the total simulation time shows a difference of a factor ~ 3 with respect to our compiler without garbage collection. However, also the heap memory management shows a difference of around 9%.

For all the models we finally measured the number of heap allocations regarding the arrays passed as input and returned as output by the functions. The allocations of the model's variables are not taken into consideration, as they live throughout the whole execution and thus are not a matter of the transformation passes described in section 3. The results are shown in table 3.

Table 3 Number of malloc calls performed by each model when compiled by OpenModelica (OM) and the prototype LLVM-based compiler.

	OpenModelica	**Prototype**	
Garbage collection	Yes	Yes	No
Scalar model	0	0	0
Fixed size array model	2000006	0	0
Dynamic size array model	2000006	2000000	2000000

The scalar case is trivial in both cases since functions deal only with scalar variables, both in input and output. The second model deals with arrays of fixed size. OpenModelica allocates all of them on the heap, while our prototype compiler takes advantage of the optimization described in section 3.1. All the output arrays are in fact promoted to arguments and thus allocated on the stack by the caller, together with the input ones. Finally, the model with dynamically sized arrays can not be optimized and thus the heap allocation persist. The slightly different number of allocations between the two compilers is given by the fact that OpenModelica also performs some additional simulation cycles for initialization purposes, whose details are not a concern of this document.

5 Conclusions

We introduced two optimization passes to improve the memory management within the Modelica simulations. The first transformation consists in analyzing the signature of each function and promoting the output arrays with fixed-size dimensions to arguments, so that the allocation is delegated to the caller and thus the stack can be used. The second aims to correctly place the deallocation instructions for the dynamically sized arrays, which are instead always placed on the heap due to their nature.

We then implemented such transformations within an LLVM-based prototype compiler and we checked their correctness by means of both an internal profiler and the external Valgrind tool. We also modified our prototype compiler to leverage the Boehm garbage collector instead of our new deallocation strategy. Even though the garbage collector manifested an efficient memory management, results showed that the time overhead is not irrelevant. Avoiding garbage collection led to a speed-up of a factor ~ 13 for the heap management and an overall 6% reduction of the total execution time.

Finally, we compared the simulations generated by our prototype compiler with the ones generated by OpenModelica. As expected, the output arrays promotion for fixed-size arrays took place and led to zero heap allocations, while OpenModelica showed the same number of allocations in both the fixed and dynamically sized arrays scenarios. For what regards the performance measurement, we focused our attention on a model with dynamically sized arrays – that is where we expected the biggest improvement. We registered a 9% reduction in the time spent in heap memory management and a speed-up of factor ~ 3 for the whole simulation.

While the work presented in this paper effectively handles memory management in Modelica compilers, there are several other key aspects for improving the performance of both the compiler and the generated code. In particular, future directions for our work aim primarily at extending and improving our prototype compiler, with the goal of efficiently handling Modelica equation arrays [1].

References

1 Giovanni Agosta, Emanuele Baldino, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a high-performance modelica compiler. In *Proceedings of the 13th International Modelica Conference*, pages 313–320, 2019. doi:10.3384/ecp19157313.

2 Giovanni Agosta, Francesco Casella, Stefano Cherubin, Alberto Leva, and Federico Terraneo. Towards a benchmark suite for high-performance Modelica compilers. In *9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, November 2019. doi:10.1145/3365984.3365988.

3 Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. Jmodelica – an open source platform for optimization of modelica models. In *6th Vienna International Conference on Mathematical Modelling*, 2009.

4 Mats Andersson. An object-oriented language for model representation. In *Proc. 2nd IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, pages 8–15, Tampa, FL, USA, 1989.

5 Barbara Rita Barricelli, Elena Casiraghi, and Daniela Fogli. A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7:167653–167671, 2019. doi:10.1109/ACCESS.2019.2953499.

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

7 Francesco Casella. Simulation of large-scale models in modelica: State of the art and future perspectives. In *11th Int'l Modelica Conference*, pages 459–468, 2015.

8 Francesco Casella, Martin Otter, Katrin Proelss, Christoph Richter, and Hubertus Tummescheit. The modelica fluid and media library for modeling of incompressible and compressible thermo-fluid pipe networks. In *Proceedings of the 5th international modelica conference*, pages 631–640, 2006.

9 Hilding Elmqvist. DYMOLA – a structured model language for large continuous systems. In *Proc. Summer Computer Simulation Conference*, Toronto, Canada, 1979.

10 Peter Fritzson, Peter Aronsson, Adrian Pop, Hakan Lundvall, Kaj Nystrom, Levon Saldamli, David Broman, and Anders Sandholm. Openmodelica-a free open-source environment for system modeling, simulation, and teaching. In *2006 IEEE Conf on Computer Aided Control System Design, 2006 IEEE Int'l Conf on Control Applications, 2006 IEEE Int'l Sym on Intelligent Control*, pages 1588–1595. IEEE, 2006.

11 Peter Fritzson, Adrian Pop, Adeel Asghar, Bernhard Bachmann, Willi Braun, Robert Braun, Lena Buffoni, Francesco Casella, Rodrigo Castro, Alejandro Danós, et al. The openmodelica integrated modeling, simulation and optimization environment. In *Proceedings of the 1st American Modelica Conference*, pages 207–220. Modelica Association, 2018.

12 John B Kam and Jeffrey D Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

13 Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

14 Oliver Lenord, Martin Otter, Christoff Bürger, Michael Hussmann, Pierre Le Bihan, Jörg Niere, Andreas Pfeiffer, Robert Reicherdt, and Kai Werther. efmi: An open standard for physical models in embedded software. In *Proceedings of 14th Modelica Conference*, 2021. doi:10.3384/ecp2118157.

15 Kendrik Yan Hong Lim, Pai Zheng, and Chun-Hsien Chen. A state-of-the-art survey of digital twin: techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing*, 31(6):1313–1337, 2020.

16 Mengnan Liu, Shuiliang Fang, Huiyue Dong, and Cunzhi Xu. Review of digital twin about concepts, technologies, and industrial applications. *Journal of Manufacturing Systems*, 58:346–361, 2021. Digital Twin towards Smart Manufacturing and Industry 4.0. doi:10.1016/j.jmsy.2020.06.017.

17 Sven Erik Mattsson, Hilding Elmqvist, and Martin Otter. Physical system modeling with modelica. *Control Engineering Practice*, 6(4):501–510, 1998.

18 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

19 Arunkumar Palanisamy, Adrian Pop, Martin Sjölund, and Peter Fritzson. Modelica based parser generator with good error handling. In *Proceedings of the 10th International Modelica Conference; March 10-12; 2014; Lund; Sweden.* number 096, pages 567–575. Linköping University Electronic Press, 2014.

20 Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, Rüdiger Franke, et al. A new openmodelica compiler high performance frontend. In *13th International Modelica Conference*, volume 157, pages 689–698, 2019.

21 Martin Sjölund, Peter Fritzson, and Adrian Pop. Bootstrapping a modelica compiler aiming at modelica 4. In *8th Int'l Modelica Conference, Dresden, Germany*, pages 510–521. Linköping University Electronic Press, 2011.

22 The Mathworks, Inc. Simscape documentation. `https://mathworks.com/help/physmod/simscape/`, 2022 (latest version).

23 John Tinnerholm. An llvm backend for the open modelica compiler, 2019.