

Simulation of Large-Scale Modelica Models with Array-Preserving Technology: Early Results and Perspectives

Francesco Casella
(francesco.casella@polimi.it)

OpenModelica

Outline

- Systems-of-Systems simulation
- Historical Perspective
- Fundamental Problems & Enabling Factors
- Array-Preserving Flattening
- Array-Preserving Structural Analysis & Optimization
- Array-Preserving Code Generation
- Sparse Solvers
- Status and Perspectives with OpenModelica
- Status and Perspectives with Other Tools
- Conclusions & Outlook

Systems-of-Systems Simulation

From Systems to Systems-of-Systems Simulation



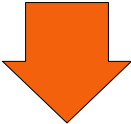
From Systems to Systems-of-Systems Simulation



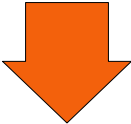
From Systems to Systems-of-Systems Simulation



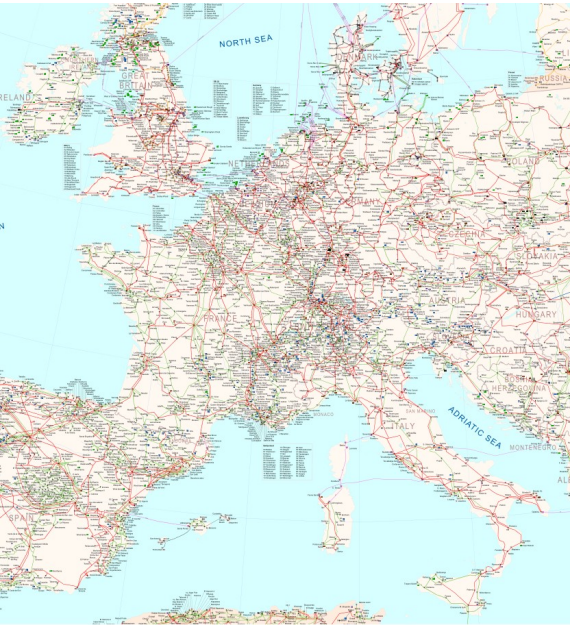
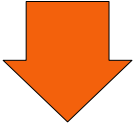
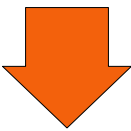
From Systems to Systems-of-Systems Simulation



From Systems to Systems-of-Systems Simulation



From Systems to Systems-of-Systems Simulation



(My) Historical Perspective

The Quest for Larger and Larger EOO Models

- 1980's: Hilding Elmqvist's PhD work (100 equations)

The Quest for Larger and Larger EOO Models

- 1980's: Hilding Elmqvist's PhD work (100 equations)
- 1990's: Early Modelica models, first multibody library, hybrid drivetrains (1000 equations)

The Quest for Larger and Larger EOO Models

- 1980's: Hilding Elmqvist's PhD work (100 equations)
- 1990's: Early Modelica models, first multibody library, hybrid drivetrains (1000 equations)
- 2000's: Thermo-Fluid models, Power Plant models (10.000 equations)

The Quest for Larger and Larger EOO Models

- 1980's: Hilding Elmqvist's PhD work (100 equations)
- 1990's: Early Modelica models, first multibody library, hybrid drivetrains (1000 equations)
- 2000's: Thermo-Fluid models, Power Plant models (10.000 equations)
- 2007 (?) F. Cellier discusses Modelica electronic circuit models (the case of the 16 bit multiplier)

The Quest for Larger and Larger EOO Models

- 1980's: Hilding Elmqvist's PhD work (100 equations)
- 1990's: Early Modelica models, first multibody library, hybrid drivetrains (1000 equations)
- 2000's: Thermo-Fluid models, Power Plant models (10.000 equations)
- 2007 (?) F. Cellier discusses Modelica electronic circuit models (the case of the 16 bit multiplier)
- 2010's: More detailed Modelica models (100.000 equations)

2015: Paper on Large-Scale Modelica Models

Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives

Francesco Casella

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
francesco.casella@polimi.it

Abstract

State-of-the-art Modelica tools are very effective at converting declarative models based on differential-algebraic equations into ordinary differential equations. However, when confronted with large-scale models of distributed systems with a high number of states (1000 or more) or with large algebraic systems of equations (1000 or more unknowns), they face a number of serious efficiency issues, that hamper their practical use for system design. The paper analyses these issues in detail, points out strategies for improvement, and also introduces a library of scalable test models that can be used to assess existing tools, as well as to help developing advanced solution methods for large-scale systems.

Keywords: Modelica Compilers, Large-Scale Models, Efficient Simulation

1 Introduction

After almost 20 years from the first release of the Modelica language definition 1.0 (The Modelica Association, 1997), the Modelica language is well-established for system-level modelling tasks in many domains of engineering, such as automotive, robotics, mechatronics, energy, aerospace, in particular when multi-domain modelling is required.

To the best of the author's knowledge, based on published literature and personal experience, the standard work flow of state-of-the art Modelica tools can be summarised by the following steps, which are described in detail by Cellier and Kofman (2006).

1. (*Flattening*) The Modelica code is parsed; classes are expanded and instantiated, and eventually brought into the so-called flat form, i.e., a set of scalar hybrid differential-algebraic equations together with a set of scalar variables and parameters.
2. (*Causalisation*) Structural analysis of the differential-algebraic equations (DAEs) is performed, in order to solve them efficiently for the state derivatives and algebraic variables. This

process includes equation ordering (BLT transformation), may require symbolic index reduction, and usually involves extensive symbolic processing, as well as the use of advanced techniques such as tearing or reshuffling for solving sub-systems of equations efficiently. In most cases, the use of numerical solvers for linear and non-linear systems of algebraic equations is required.

3. (*Time integration*) The code which results from the previous step is linked to some well-tested, general-purpose dense Ordinary Differential Equation (ODE) solver, including root-finding algorithms to handle state events in the case of hybrid models.

In principle, step 2 is not strictly necessary, as DAEs resulting from step 1 could be solved directly using numerical DAE solvers. In practice, this is not standard practice for two reasons: one is that object-oriented Modelica models very often end up having index greater than 1, that are challenging to solve numerically, the other is that the above-sketched process is usually more numerically robust and easier to initialize than the direct solution of the nonlinear DAEs.

As to step 3, most Modelica models end up being stiff, because the modular way of building the models very often generates some very fast dynamic phenomena that, albeit maybe not of interest for the modeller, cannot be easily removed from the model, because they stem from the interaction of equations placed in different components.

As a consequence, stiff solvers are usually needed, the choice usually falling onto DASSL (for multi-step algorithms) and on Radau IIa (for single-step algorithms), which implement sophisticated step-size and order adaptation with error control, as well as root-finding algorithms for state-event detection.

When explicit solvers are required (e.g., for real-time simulation applications) it is sometimes possible to carefully build a modular model so that stiffness is avoided, but this is not the standard way people build object-oriented models in most cases, and people usually take for granted that stiffness will be handled by the solver.

- Introduces ScalableTestSuite library
- Points out need for:
 - Sparse solvers
 - Multi-rate algorithms
 - Multi-rate event-handling
 - QSS algorithms
 - Exploiting repetitive structures
 - Exploiting parallel CPUs

2015: Paper on Large-Scale Modelica Models

Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives

Francesco Casella

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
francesco.casella@polimi.it

Abstract

State-of-the-art Modelica tools are very effective at converting declarative models based on differential-algebraic equations into ordinary differential equations. However, when confronted with large-scale models of distributed systems with a high number of states (1000 or more) or with large algebraic systems of equations (1000 or more unknowns), they face a number of serious efficiency issues, that hamper their practical use for system design. The paper analyses these issues in detail, points out strategies for improvement, and also introduces a library of scalable test models that can be used to assess existing tools, as well as to help developing advanced solution methods for large-scale systems.

Keywords: Modelica Compilers, Large-Scale Models, Efficient Simulation

1 Introduction

After almost 20 years from the first release of the Modelica language definition 1.0 (The Modelica Association, 1997), the Modelica language is well-established for system-level modelling tasks in many domains of engineering, such as automotive, robotics, mechatronics, energy, aerospace, in particular when multi-domain modelling is required.

To the best of the author's knowledge, based on published literature and personal experience, the standard work flow of state-of-the art Modelica tools can be summarised by the following steps, which are described in detail by Cellier and Kofman (2006).

1. (*Flattening*) The Modelica code is parsed; classes are expanded and instantiated, and eventually brought into the so-called flat form, i.e., a set of scalar hybrid differential-algebraic equations together with a set of scalar variables and parameters.
2. (*Causalisation*) Structural analysis of the differential-algebraic equations (DAEs) is performed, in order to solve them efficiently for the state derivatives and algebraic variables. This

process includes equation ordering (BLT transformation), may require symbolic index reduction, and usually involves extensive symbolic processing, as well as the use of advanced techniques such as tearing or reshuffling for solving sub-systems of equations efficiently. In most cases, the use of numerical solvers for linear and non-linear systems of algebraic equations is required.

3. (*Time integration*) The code which results from the previous step is linked to some well-tested, general-purpose dense Ordinary Differential Equation (ODE) solver, including root-finding algorithms to handle state events in the case of hybrid models.

In principle, step 2 is not strictly necessary, as DAEs resulting from step 1 could be solved directly using numerical DAE solvers. In practice, this is not standard practice for two reasons: one is that object-oriented Modelica models very often end up having index greater than 1, that are challenging to solve numerically, the other is that the above-sketched process is usually more numerically robust and easier to initialize than the direct solution of the nonlinear DAEs.

As to step 3, most Modelica models end up being stiff, because the modular way of building the models very often generates some very fast dynamic phenomena that, albeit maybe not of interest for the modeller, cannot be easily removed from the model, because they stem from the interaction of equations placed in different components.

As a consequence, stiff solvers are usually needed, the choice usually falling onto DASSL (for multi-step algorithms) and on Radau IIa (for single-step algorithms), which implement sophisticated step-size and order adaptation with error control, as well as root-finding algorithms for state-event detection.

When explicit solvers are required (e.g., for real-time simulation applications) it is sometimes possible to carefully build a modular model so that stiffness is avoided, but this is not the standard way people build object-oriented models in most cases, and people usually take for granted that stiffness will be handled by the solver.

- Introduces ScalableTestSuite library
- Points out need for:
 - Sparse solvers
 - Multi-rate algorithms
 - Multi-rate event-handling
 - QSS algorithms
 - Exploiting repetitive structures
 - Exploiting parallel CPUs

2016-2017 Sparse Solvers introduced in OMC

Solving Large-scale Modelica Models: New Approaches and Experimental Results using OpenModelica

Willi Braun¹ Francesco Casella² Bernhard Bachmann¹

¹FH Bielefeld, Bielefeld, Germany, (willi.braun,bernhard.bachmann}@fh-bielefeld.org

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy,
francesco.casella@polimi.it

Abstract

Modelica-based modeling and simulation is becoming increasingly important for the development of high quality engineering products. Therefore, the system size of interest in a Modelica-based simulation is continuously increasing and the traditional way of generating simulation code, e.g. involving symbolic transformations like matching, sorting, and tearing, must be adapted to this situation. This paper describes recently implemented sparse solver techniques in OpenModelica in order to efficiently compile and simulate large-scale Modelica models. A proof of concept is given by evaluating the performance of selected benchmark problems.

Keywords: Modelica, large-scale, sparse solver techniques

1 Introduction

The design and safe operation of modern large-scale cyber-physical systems requires the ability to model and simulate them efficiently. The Modelica language is optimally suited for the modelling task, thanks to the high-level declarative modelling approach and to the powerful object-oriented features such as inheritance and replaceable objects. On the other hand, as noted in (Casella, 2015), until recently the development of Modelica tools has been focused on the modelling of moderate-sized models, optimizing the simulation code as much as possible by means of structural analysis and symbolic processing of the system of equations.

Large system models are usually characterized by a high degree of sparsity, since each component interacts only with a few neighbours, so that each differential-algebraic equation in the model only depends on a handful of variables. The availability of reliable open-source sparse solvers (Hindmarsh et al., 2005; Davis and Natarajan, 2010) and of cheap computing power and memory even on low-end workstations opens up the possibility of tackling much large system models, featuring hundreds of thousands or possibly millions of equations, exploiting the sparsity of such models for their solution.

In particular, the interest in the use of Modelica for the modelling and simulation of national- and continental-sized power generation and transmission systems recently

motivated a first exploratory effort in this direction, using OpenModelica as a development platform, see (Casella et al., 2016). The methods implemented for the power system studies also allowed to efficiently simulate the cooling blanket of the future DEMO nuclear fusion reactor, which requires the modelling of thousands of individual heat-exchanging pipes, see (Froio et al., 2016).

The goal of this paper is threefold: to discuss different strategies for the simulation of large-scale Modelica models using sparse solvers; to describe an implementation of such strategies in the OpenModelica Compiler (OMC), using open-source solvers; finally, to present and discuss the performance obtained in a number of benchmark cases. The numerical methods are discussed in Section 2. The simulation performance is analyzed on three sets of benchmarks: the ScalableTestSuite library (Casella, 2015; Casella and Sezginer, 2016), some large power system models (Casella et al., 2016), and large high-fidelity models of the cooling system of the future DEMO nuclear fusion plant (Froio et al., 2017); results are reported in Section 3. Finally, Section 4 concludes the paper and gives an outlook to future work.

2 Solving Modelica Models

2.1 ODE mode

2.1.1 Symbolic Transformation Steps

In common Modelica tools the compile process can be summarized with the following steps, which are also explained in (Cellier and Kofman, 2006):

Flattening The Modelica model is transformed by the front-end into a flat representation, consisting essentially of lists of variables, functions, equations and algorithms.

Pre-Optimization In this phase a basic structural analysis of the differential-algebraic equations (DAE) is performed, e.g. detecting the potential states and discrete variables, eliminating alias variables.

Causalization This is a basic step in a Modelica Compiler, the so-called BLT-Transformation. Matching, sorting, and index reduction algorithms are applied

- Sparse solvers for implicit ODE integration (IDA)
- Sparse solvers for implicit algebraic loops (Kinsol/KLU)
- Sparse solvers for daeMode integration (IDA)

2016-2017 Sparse Solvers introduced in OMC

Solving Large-scale Modelica Models: New Approaches and Experimental Results using OpenModelica

Willi Braun¹ Francesco Casella² Bernhard Bachmann¹

¹FH Bielefeld, Bielefeld, Germany, (willi.braun,bernhard.bachmann}@fh-bielefeld.org

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy,
francesco.casella@polimi.it

Abstract

Modelica-based modeling and simulation is becoming increasingly important for the development of high quality engineering products. Therefore, the system size of interest in a Modelica-based simulation is continuously increasing and the traditional way of generating simulation code, e.g. involving symbolic transformations like matching, sorting, and tearing, must be adapted to this situation. This paper describes recently implemented sparse solver techniques in OpenModelica in order to efficiently compile and simulate large-scale Modelica models. A proof of concept is given by evaluating the performance of selected benchmark problems.

Keywords: Modelica, large-scale, sparse solver techniques

1 Introduction

The design and safe operation of modern large-scale cyber-physical systems requires the ability to model and simulate them efficiently. The Modelica language is optimally suited for the modelling task, thanks to the high-level declarative modelling approach and to the powerful object-oriented features such as inheritance and replaceable objects. On the other hand, as noted in (Casella, 2015), until recently the development of Modelica tools has been focused on the modelling of moderate-sized models, optimizing the simulation code as much as possible by means of structural analysis and symbolic processing of the system of equations.

Large system models are usually characterized by a high degree of sparsity, since each component interacts only with a few neighbours, so that each differential-algebraic equation in the model only depends on a handful of variables. The availability of reliable open-source sparse solvers (Hindmarsh et al., 2005; Davis and Natarajan, 2010) and of cheap computing power and memory even on low-end workstations opens up the possibility of tackling much large system models, featuring hundreds of thousands or possibly millions of equations, exploiting the sparsity of such models for their solution.

In particular, the interest in the use of Modelica for the modelling and simulation of national- and continental-sized power generation and transmission systems recently

motivated a first exploratory effort in this direction, using OpenModelica as a development platform, see (Casella et al., 2016). The methods implemented for the power system studies also allowed to efficiently simulate the cooling blanket of the future DEMO nuclear fusion reactor, which requires the modelling of thousands of individual heat-exchanging pipes, see (Froio et al., 2016).

The goal of this paper is threefold: to discuss different strategies for the simulation of large-scale Modelica models using sparse solvers; to describe an implementation of such strategies in the OpenModelica Compiler (OMC), using open-source solvers; finally, to present and discuss the performance obtained in a number of benchmark cases. The numerical methods are discussed in Section 2. The simulation performance is analyzed on three sets of benchmarks: the ScalableTestSuite library (Casella, 2015; Casella and Sezginer, 2016), some large power system models (Casella et al., 2016), and large high-fidelity models of the cooling system of the future DEMO nuclear fusion plant (Froio et al., 2017); results are reported in Section 3. Finally, Section 4 concludes the paper and gives an outlook to future work.

2 Solving Modelica Models

2.1 ODE mode

2.1.1 Symbolic Transformation Steps

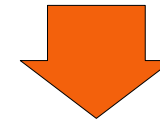
In common Modelica tools the compile process can be summarized with the following steps, which are also explained in (Cellier and Kofman, 2006):

Flattening The Modelica model is transformed by the front-end into a flat representation, consisting essentially of lists of variables, functions, equations and algorithms.

Pre-Optimization In this phase a basic structural analysis of the differential-algebraic equations (DAE) is performed, e.g. detecting the potential states and discrete variables, eliminating alias variables.

Causalization This is a basic step in a Modelica Compiler, the so-called BLT-Transformation. Matching, sorting, and index reduction algorithms are applied

- Sparse solvers for implicit ODE integration (IDA)
- Sparse solvers for implicit algebraic loops (Kinsol/KLU)
- Sparse solvers for daeMode integration (IDA)



- Simulation of systems with 1M equations becomes possible in OMC

2016-2017 Early Experiments with Large Models in OMC

Simulation of Large Grids in OpenModelica: reflections and perspectives

Francesco Casella¹ Alberto Leva¹ Andrea Bartolini²

¹Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy.

{francesco.casella,alberto.leva}@polimi.it

²Dynamica s.r.l., Italy, andrea.bartolini@dynamica-it.com

Abstract

This paper belongs to a long-term research activity on modelling and simulation of large-size power grids in Modelica, using the OpenModelica Compiler. We describe the present state of the research, its evolution over the last year, the conclusions we could reach in this period in comparison with the initial hypotheses, and some results. Finally, we outline the future of the presented activity.

Keywords: Grid Modelling and Simulation, Large-Scale Systems, Efficient Simulation.

1 Introduction

The modelling and simulation of large power grids is an emerging domain of interest for the Modelica language, as the encountered problems basically consist of large networked systems with decentralized control, where multiple producers and consumers cooperate to the goals of stable network behaviour, satisfaction of all the load requests, and system optimality.

Although control strategies for such large-scale systems are usually designed as hierarchical systems, abstracting low-level behaviours within higher levels, it is sometimes necessary to simulate the entire system. This can be the case when a full verification of the designed strategy, including the interactions among its parts, is in order—and this is an issue shared by any large-scale system.

In the case of electric grids, there is another problem to address. For management reasons at the nation- or continent-wide scale, it is required to periodically assemble a model of the entire system and use it to run numerous simulations, to verify that the stress expected in the next time period can be sustained without incurring in stability problems, to test critical manoeuvres when required, and possibly to take decisions in a view to optimise the operation. This particular use of simulation makes a fast code generation vital.

Over the last two years, we have been working on this subject, with the goal of providing an entirely Modelica-based solution using the open-source OpenModelica Compiler (OMC) for code generation. The problem at hand is one very interesting case of an emerging class of large-scale models, see (Casella, 2015) for an

overall discussion on this topic. Preliminary results were presented in (Casella et al., 2016), which was mainly addressed to the power system community. This paper incorporates the results of additional work carried out since then, and presents the current state of the research from the perspective of the Modelica community.

2 Previous research

In this section we summarise the research context and the results from which we started, referring the interested reader to (Casella et al., 2016) for further details.

National grids in Europe are rapidly evolving (ENTSO-E, 2015, 2014). The penetration of intermittent sources like wind and solar enhances the need for continent-level integration for countries to help one another. Transmission networks are moving from the traditional structure dominated by large synchronous generators and AC links, toward an increasing share of HVDC links and of medium- and small-scale generators interfaced to the grid via AC/DC/AC links. As a consequence, the management of transmission grids by national Transmission System Operators (TSOs) increasingly requires knowledge of the dynamic behaviour of the the system outside the country boundaries.

Traditionally, well-established domain-specific tools are used such as PowerFactory, PSS/E, and Eurostag. These tools come with extensive component libraries, but the exact formulation of the said models is difficult to access, since they are written in low-level languages like FORTRAN. With commercial tools, the models' source code might even be unavailable to the end user. This hinders the required interoperability, as models of the same object in different tools may behave differently. Indeed, full interoperability would ideally require all European TSOs to use the same simulation tool.

Modelica has been already used for the modelling of electrical power systems, including detailed machine models (Franke and Wiesmann, 2014; Kral and Haumer, 2005), and more recently it has been considered also to model electro-mechanical transients in high-voltage generation and transmission system. In this context, an activity worth mentioning is the iTesla European FP7 research project (Vanfretti et al., 2013, 2014; Zhang et al., 2015), although the results of the project refer to small- or

- Successful simulation of power grid models up to 600.000 equations
- Model build time: 15+ minutes
- Model simulation time: 3 minutes
- Required memory: 72 GB
- Simulation executable size: ~1 GB

Network	Nodes	Gens	Lines	Trafos	Equations
GRID_C	751	74	369	583	56386
GRID_E	1817	267	1458	1202	157022
GRID_D	8376	2317	1946	2489	579470
GRID_G	8113	407	6833	2824	593886

Network	Flattening	C gen.	Compilation	Simulation
GRID_C	24	24	13	12
GRID_E	73	67	35	44
GRID_D	334	315	123	111
GRID_G	318	303	144	186

2016-2017 Early Experiments with Large Models in OMC

Simulation of Large Grids in OpenModelica: reflections and perspectives

Francesco Casella¹ Alberto Leva¹ Andrea Bartolini²

¹Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy.

{francesco.casella,alberto.leva}@polimi.it

²Dynamica s.r.l., Italy, andrea.bartolini@dynamica-it.com

Abstract

This paper belongs to a long-term research activity on modelling and simulation of large-size power grids in Modelica, using the OpenModelica Compiler. We describe the present state of the research, its evolution over the last year, the conclusions we could reach in this period in comparison with the initial hypotheses, and some results. Finally, we outline the future of the presented activity.

Keywords: Grid Modelling and Simulation, Large-Scale Systems, Efficient Simulation.

1 Introduction

The modelling and simulation of large power grids is an emerging domain of interest for the Modelica language, as the encountered problems basically consist of large networked systems with decentralized control, where multiple producers and consumers cooperate to the goals of stable network behaviour, satisfaction of all the load requests, and system optimality.

Although control strategies for such large-scale systems are usually designed as hierarchical systems, abstracting low-level behaviours within higher levels, it is sometimes necessary to simulate the entire system. This can be the case when a full verification of the designed strategy, including the interactions among its parts, is in order—and this is an issue shared by any large-scale system.

In the case of electric grids, there is another problem to address. For management reasons at the nation- or continent-wide scale, it is required to periodically assemble a model of the entire system and use it to run numerous simulations, to verify that the stress expected in the next time period can be sustained without incurring in stability problems, to test critical manoeuvres when required, and possibly to take decisions in a view to optimise the operation. This particular use of simulation makes a fast code generation vital.

Over the last two years, we have been working on this subject, with the goal of providing an entirely Modelica-based solution using the open-source OpenModelica Compiler (OMC) for code generation. The problem at hand is one very interesting case of an emerging class of large-scale models, see (Casella, 2015) for an

overall discussion on this topic. Preliminary results were presented in (Casella et al., 2016), which was mainly addressed to the power system community. This paper incorporates the results of additional work carried out since then, and presents the current state of the research from the perspective of the Modelica community.

2 Previous research

In this section we summarise the research context and the results from which we started, referring the interested reader to (Casella et al., 2016) for further details.

National grids in Europe are rapidly evolving (ENTSO-E, 2015, 2014). The penetration of intermittent sources like wind and solar enhances the need for continent-level integration for countries to help one another. Transmission networks are moving from the traditional structure dominated by large synchronous generators and AC links, toward an increasing share of HVDC links and of medium- and small-scale generators interfaced to the grid via AC/DC/AC links. As a consequence, the management of transmission grids by national Transmission System Operators (TSOs) increasingly requires knowledge of the dynamic behaviour of the the system outside the country boundaries.

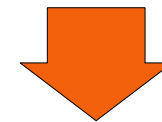
Traditionally, well-established domain-specific tools are used such as PowerFactory, PSS/E, and Eurostag. These tools come with extensive component libraries, but the exact formulation of the said models is difficult to access, since they are written in low-level languages like FORTRAN. With commercial tools, the models' source code might even be unavailable to the end user. This hinders the required interoperability, as models of the same object in different tools may behave differently. Indeed, full interoperability would ideally require all European TSOs to use the same simulation tool.

Modelica has been already used for the modelling of electrical power systems, including detailed machine models (Franke and Wiesmann, 2014; Kral and Haumer, 2005), and more recently it has been considered also to model electro-mechanical transients in high-voltage generation and transmission system. In this context, an activity worth mentioning is the iTesla European FP7 research project (Vanfretti et al., 2013, 2014; Zhang et al., 2015), although the results of the project refer to small- or

- Successful simulation of power grid models up to 600.000 equations
- Model build time: 15+ minutes
- Model simulation time: 3 minutes
- Required memory: 72 GB
- Simulation executable size: ~1 GB

Network	Nodes	Gens	Lines	Trafos	Equations
GRID_C	751	74	369	583	56386
GRID_E	1817	267	1458	1202	157022
GRID_D	8376	2317	1946	2489	579470
GRID_G	8113	407	6833	2824	593886

Network	Flattening	C gen.	Compilation	Simulation
GRID_C	24	24	13	12
GRID_E	73	67	35	44
GRID_D	334	315	123	111
GRID_G	318	303	144	186



- Simulation performance OK (SoA sparse solver)
- Build time way too long for the user's requirements

2016-2017 Early Experiments with Large Models in OMC

Fusion Engineering and Design 124 (2017) 887–891



Contents lists available at ScienceDirect

Fusion Engineering and Design

journal homepage: www.elsevier.com/locate/fusengdes



Dynamic thermal-hydraulic modelling of the EU DEMO WCLL breeding blanket cooling loops



A. Froio^a, F. Casella^b, F. Cismondi^c, A. Del Nevo^d, L. Savoldi^a, R. Zanino^{d,*}

^a NEMO Group, Dipartimento Energia, Politecnico di Torino, 10129 Torino, Italy, Italy

^b Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milano, Italy

^c PPPT Department, EUROfusion Consortium, 85748 Garching bei München, Germany

^d ENEA CR Brasimone, 40032 Camugnano, BO, Italy

HIGHLIGHTS

888

A. Froio et al. / Fusion Engineering and Design 124 (2017) 887–891

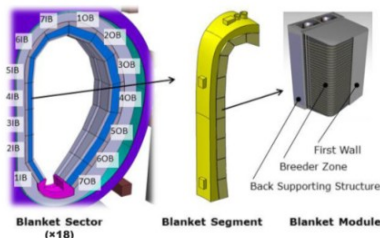


Fig. 1. Blanket segmentation, showing the numbering convention of the IB1-7 and OB1-7 BMs (reproduced from [5]).

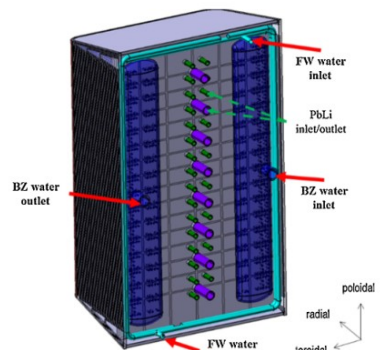


Fig. 2. CAD of the rear side of the WCLL OB4 BM, showing the coolant I/O manifolds.

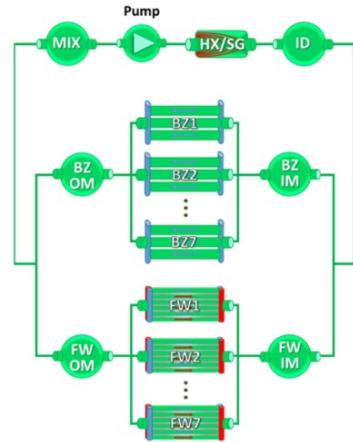


Fig. 3. Schematic of the WCLL cooling circuit model (FW#: First Wall object; BZ#: Breeding Zone object; IM: Inlet Manifold; OM: Outlet Manifold; ID: Inlet Distributor; MIX: Mixer; HX/SG: Heat exchanger/Steam Generator).

is included in the FW cooling circuit model by treating the channel walls as 1D objects in the flow direction, see [2] for details.

The BZ part, whose model is shown in Fig. 5, is cooled by circular double-wall tubes, which are in contact with the PbLi flowing in the free space on their outer side. The tubes are arranged in a modular layout, with a set of elementary cells of 21 tubes (shown in Fig. 6) ideally stacked in the poloidal direction, with inlet orifices to control the mass flow rate distribution.

The primary heat sink is the heat exchanger (HX), which can be a steam generator (SG), if the primary heat is to be directly used to

- Detail thermal model of DEMO fusion reactor blanket circuit (750.000 equations)
- Build once, run many times
- Good for optimization purposes

2016-2017 Early Experiments with Large Models in OMC

Fusion Engineering and Design 124 (2017) 887–891



Contents lists available at ScienceDirect
Fusion Engineering and Design
 journal homepage: www.elsevier.com/locate/fusengdes



Dynamic thermal-hydraulic modelling of the EU DEMO WCLL breeding blanket cooling loops



A. Froio^a, F. Casella^b, F. Cisondi^c, A. Del Nevo^d, L. Savoldi^a, R. Zanino^{d,*}

^a NEMO Group, Dipartimento Energia, Politecnico di Torino, 10129 Torino, Italy, Italy

^b Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, 20133 Milano, Italy

^c PPPT Department, EUROfusion Consortium, 85748 Garching bei München, Germany

^d ENEA CR Brasimone, 40032 Camugnano, BO, Italy

HIGHLIGHTS

888

A. Froio et al. / Fusion Engineering and Design 124 (2017) 887–891

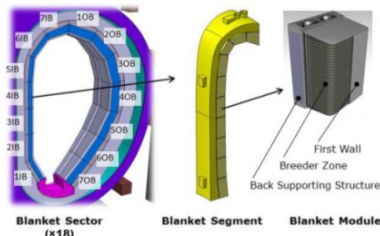


Fig. 1. Blanket segmentation, showing the numbering convention of the IB1-7 and OB1-7 BMs (reproduced from [5]).

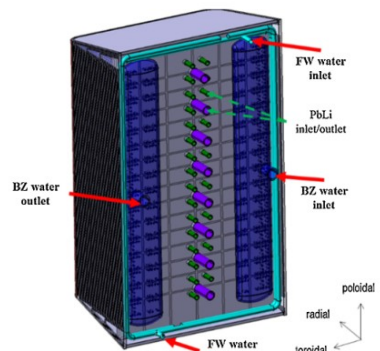


Fig. 2. CAD of the rear side of the WCLL OB4 BM, showing the coolant I/O manifolds.

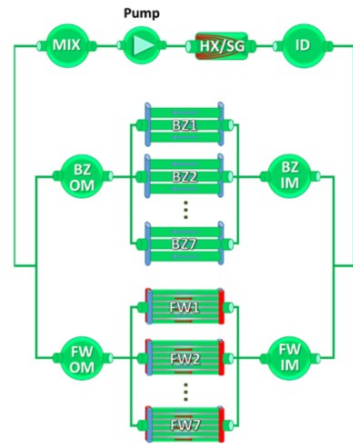


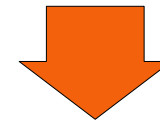
Fig. 3. Schematic of the WCLL cooling circuit model (FW#: First Wall object; BZ#: Breeding Zone object; IM: Inlet Manifold; OM: Outlet Manifold; ID: Inlet Distributor; MIX: Mixer; HX/SG: Heat exchanger/Steam Generator).

is included in the FW cooling circuit model by treating the channel walls as 1D objects in the flow direction, see [2] for details.

The BZ part, whose model is shown in Fig. 5, is cooled by circular double-wall tubes, which are in contact with the PbLi flowing in the free space on their outer side. The tubes are arranged in a modular layout, with a set of elementary cells of 21 tubes (shown in Fig. 6) ideally stacked in the poloidal direction, with inlet orifices to control the mass flow rate distribution.

The primary heat sink is the heat exchanger (HX), which can be a steam generator (SG), if the primary heat is to be directly used to

- Detail thermal model of DEMO fusion reactor blanket circuit (750.000 equations)
- Build once, run many times
- Good for optimization purposes



- Simulation performance OK (SoA sparse solver)
- Very long build time irrelevant in this case

***Fundamental
Problems
&
Enabling
Factors***

Fundamental Problems in Large-Scale Modelica Models

- The Modelica Specification describes flattening as the reduction of a Modelica model to *scalar* equations and variables
 - Doable w/o problems up to 100.000-200.000 equations
 - Doable but problematic up to 1.000.000 equations
 - Impractical above 1.000.000 equations

Fundamental Problems in Large-Scale Modelica Models

- The Modelica Specification describes flattening as the reduction of a Modelica model to *scalar* equations and variables
 - Doable w/o problems up to 100.000-200.000 equations
 - Doable but problematic up to 1.000.000 equations
 - Impractical above 1.000.000 equations
- Systems-of-systems models contain many instances of the same model and/or large arrays of models and/or large variables arrays within models

Fundamental Problems in Large-Scale Modelica Models

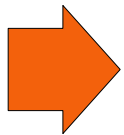
- The Modelica Specification describes flattening as the reduction of a Modelica model to *scalar* equations and variables
 - Doable w/o problems up to 100.000-200.000 equations
 - Doable but problematic up to 1.000.000 equations
 - Impractical above 1.000.000 equations
- Systems-of-systems models contain many instances of the same model and/or large arrays of models and/or large variables arrays within models
- Fundamental problems when flattening to scalars:
 - Large amounts of repeated generated code
 - Model building process becomes very time-consuming
 - Very large size of simulation executable, needs to be read from RAM to cache multiple times per step → memory bottleneck

Enabling Factors

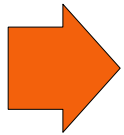
- Array-Preserving Flattening
- Array-Preserving Structural Analysis and Optimization
- Array-Preserving Code Generation
- Sparse Solvers

Enabling Factors

- Array-Preserving Flattening
- Array-Preserving Structural Analysis and Optimization
- Array-Preserving Code Generation
- Sparse Solvers



Status & Perspectives with OpenModelica



Status & Perspectives with other tools

Array-Preserving Flattening: the New Frontend (2016-2021)

A New OpenModelica Compiler High Performance Frontend

A New OpenModelica Compiler High Performance Frontend

Adrian Pop¹ Per Östlund¹ Francesco Casella² Martin Sjölund¹ Rüdiger Franke³

¹PELAB - Programming Environments Lab, Dept. of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, {adrian.pop, per.ostlund, martin.sjолund}@liu.se

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy, francesco.casella@polimi.it

³ABB, IAPG-A26, Kallstadter Str. 1, 68309 Mannheim, Germany, ruediger.franke@de.abb.com

Abstract

The equation-based object-oriented Modelica language allows easy composition of models from components. It is very easy to create very large parametrized models using component arrays of models. Current open-source and commercial Modelica tools can with ease handle models with a hundred thousand equations and a thousand states. However, when the system size goes above half a million (or more) equations the tools begin to have problems with scalability. This paper presents the new frontend of the OpenModelica compiler, designed with scalability in mind. The new OpenModelica frontend can handle much larger systems than the current one with better time and memory performance. The new frontend was validated against large models from the ScalableTestSuite library and Modelica Standard Library, with good results.

Keywords: OpenModelica, compiler, flattening, frontend, modelling, simulation, equation-based, scalability

1 Introduction and Motivation

System-level dynamic modelling and simulation is a key activity in modern system engineering design. In parallel to the detailed component design, which is performed using advanced 3D CAD, CFD and FEM software tools, system-level modelling, usually including systems of systems and large numbers of interacting components, allows predicting the dynamic performance of complex systems, which emerges from the interaction of its components.

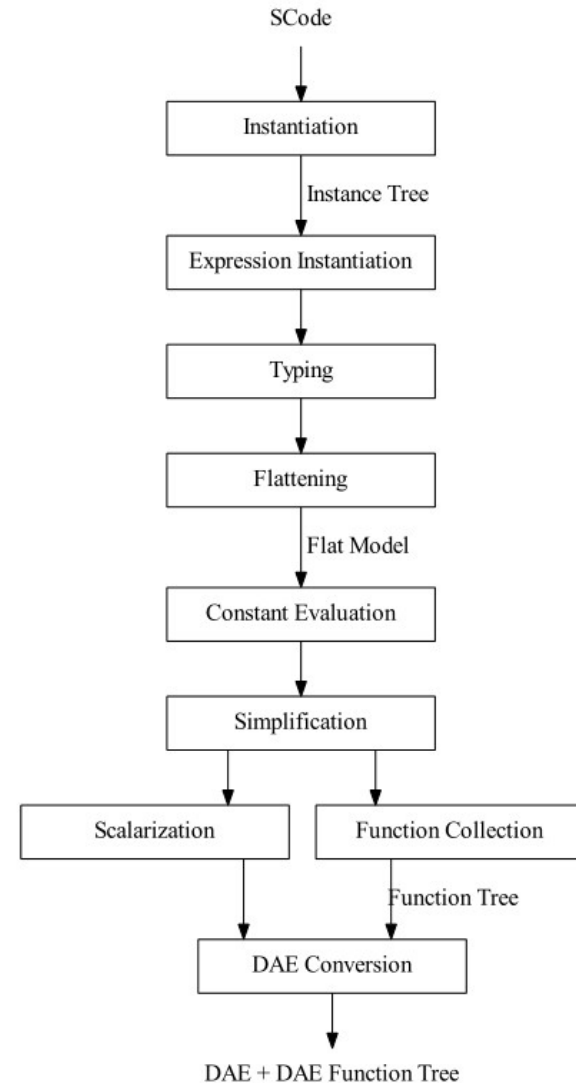
The Modelica language (Modelica Association, 2017; Fritzson, 2015) is a standardized tool-independent non-proprietary equation-based object-oriented modeling language, which was introduced 20 years ago by the non-profit Modelica Association, with strong links to industry and academia. This language, and the related eco-system of tools, model libraries and the FMI standard (Blochwitz et al., 2011), is ideally suited to system-level modeling of complex, heterogenous and multi-domain cyber-physical systems. It has become a de-facto standard in many industries, most notably the automotive one. The Modelica language is currently supported by about 10 different modeling and simulation software tools; one of them, in particular, the open-source OpenModelica software suite (Fritzson et al., 2018), is the only Modelica tool owned

and maintained by a non-profit organization – the Open Source Modelica Consortium (OSMC).

The main applications of Modelica tools so far have been the study of individual systems, such as a car's drivetrain and active suspension and steering control system, a single industrial robot, a single power plant, a single HVDC power link, the air conditioning system of a car, etc. Existing Modelica tools employ strategies and algorithms that are optimized for such system models, whose typical complexity lies in the range of 1000-50000 equations and up to a few thousand state variables. The advent of the internet-of-things paradigm is now fostering the development of innovative very large-scale cyber-physical systems, for example smart grids, or fleets of autonomous vehicles. It is also sparking a renewed interest at the modernization of traditional large-scale systems. A first example is continental-size high-voltage power generation and transmission, which is facing increasing challenges due to the introduction of power electronics equipment and to the increased penetration of intermittent renewable energy sources. A second example is district heating, possibly integrated with heat pumps and distributed power generation in an integrated electrical and thermal smart grid. See (Casella, 2015) for further examples and motivation.

Unfortunately, when Modelica is used to tackle the modelling of large-scale systems with sizes exceeding the ones mentioned above, currently available simulation software that support Modelica fall short at providing adequate performance. The time required to compile the models vastly exceeds what end users typically expect for system level studies, i.e., a few minutes at most. The size of the generated code and the memory requirements for compilers vastly exceed what is normally available on laptops and workstations used for daily work (8-16 GB).

In the last couple of years there have been some pioneering attempts at pushing the boundary of the size of Modelica models that can be handled with reasonable time and effort. In particular, some of our published papers have demonstrated the feasibility of Modelica models of high-voltage power generation and transmission systems (Braun et al., 2017; Casella et al., 2017) and of detailed models of key system components of future nuclear fusion reactors, see (Froio et al., 2017). The size of the largest models handled so far is about 750000 equations, which



Array-Preserving Flattening: the New Frontend (2016-2021)

A New OpenModelica Compiler High Performance Frontend

A New OpenModelica Compiler High Performance Frontend

Adrian Pop¹ Per Östlund¹ Francesco Casella² Martin Sjölund¹ Rüdiger Franke³

¹PELAB - Programming Environments Lab, Dept. of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, {adrian.pop, per.ostlund, martin.sjолund}@liu.se

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy, francesco.casella@polimi.it

³ABB, IAPG-A26, Kallstadter Str. 1, 68309 Mannheim, Germany, ruediger.franke@de.abb.com

Abstract

The equation-based object-oriented Modelica language allows easy composition of models from components. It is very easy to create very large parametrized models using component arrays of models. Current open-source and commercial Modelica tools can with ease handle models with a hundred thousand equations and a thousand states. However, when the system size goes above half a million (or more) equations the tools begin to have problems with scalability. This paper presents the new frontend of the OpenModelica compiler, designed with scalability in mind. The new OpenModelica frontend can handle much larger systems than the current one with better time and memory performance. The new frontend was validated against large models from the ScalableTestSuite library and Modelica Standard Library, with good results.

Keywords: OpenModelica, compiler, flattening, frontend, modelling, simulation, equation-based, scalability

1 Introduction and Motivation

System-level dynamic modelling and simulation is a key activity in modern system engineering design. In parallel to the detailed component design, which is performed using advanced 3D CAD, CFD and FEM software tools, system-level modelling, usually including systems of systems and large numbers of interacting components, allows predicting the dynamic performance of complex systems, which emerges from the interaction of its components.

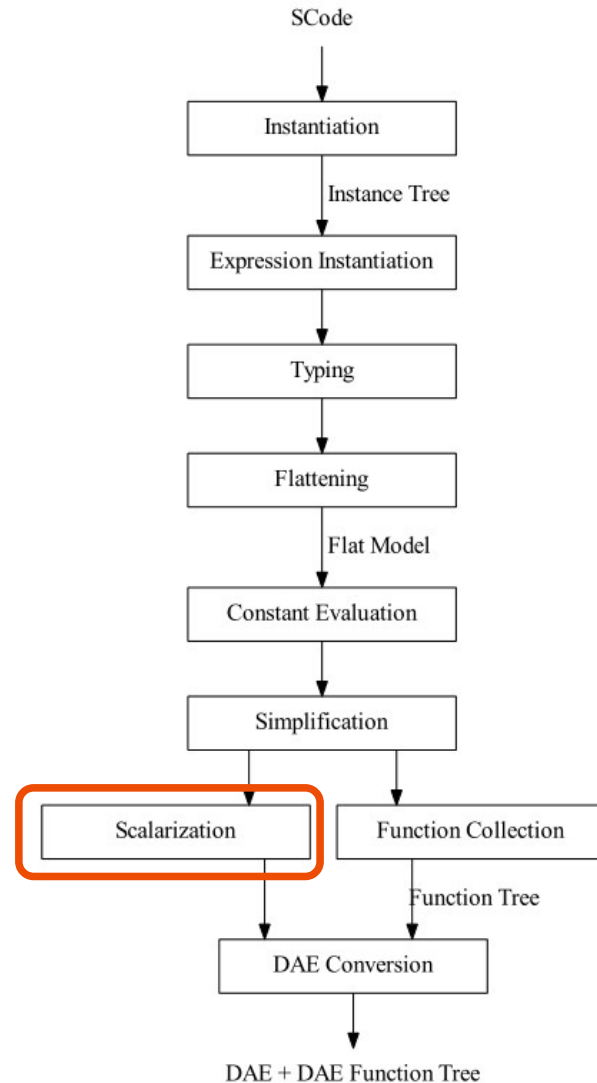
The Modelica language (Modelica Association, 2017; Fritzson, 2015) is a standardized tool-independent non-proprietary equation-based object-oriented modeling language, which was introduced 20 years ago by the non-profit Modelica Association, with strong links to industry and academia. This language, and the related eco-system of tools, model libraries and the FMI standard (Blochwitz et al., 2011), is ideally suited to system-level modeling of complex, heterogenous and multi-domain cyber-physical systems. It has become a de-facto standard in many industries, most notably the automotive one. The Modelica language is currently supported by about 10 different modeling and simulation software tools; one of them, in particular, the open-source OpenModelica software suite (Fritzson et al., 2018), is the only Modelica tool owned

and maintained by a non-profit organization – the Open Source Modelica Consortium (OSMC).

The main applications of Modelica tools so far have been the study of individual systems, such as a car's drivetrain and active suspension and steering control system, a single industrial robot, a single power plant, a single HVDC power link, the air conditioning system of a car, etc. Existing Modelica tools employ strategies and algorithms that are optimized for such system models, whose typical complexity lies in the range of 1000-50000 equations and up to a few thousand state variables. The advent of the internet-of-things paradigm is now fostering the development of innovative very large-scale cyber-physical systems, for example smart grids, or fleets of autonomous vehicles. It is also sparking a renewed interest at the modernization of traditional large-scale systems. A first example is continental-size high-voltage power generation and transmission, which is facing increasing challenges due to the introduction of power electronics equipment and to the increased penetration of intermittent renewable energy sources. A second example is district heating, possibly integrated with heat pumps and distributed power generation in an integrated electrical and thermal smart grid. See (Casella, 2015) for further examples and motivation.

Unfortunately, when Modelica is used to tackle the modelling of large-scale systems with sizes exceeding the ones mentioned above, currently available simulation software that support Modelica fall short at providing adequate performance. The time required to compile the models vastly exceeds what end users typically expect for system level studies, i.e., a few minutes at most. The size of the generated code and the memory requirements for compilers vastly exceed what is normally available on laptops and workstations used for daily work (8-16 GB).

In the last couple of years there have been some pioneering attempts at pushing the boundary of the size of Modelica models that can be handled with reasonable time and effort. In particular, some of our published papers have demonstrated the feasibility of Modelica models of high-voltage power generation and transmission systems (Braun et al., 2017; Casella et al., 2017) and of detailed models of key system components of future nuclear fusion reactors, see (Froio et al., 2017). The size of the largest models handled so far is about 750000 equations, which



Array-Preserving Flattening: the New Frontend (2016-2021)

A New OpenModelica Compiler High Performance Frontend

A New OpenModelica Compiler High Performance Frontend

Adrian Pop¹ Per Östlund¹ Francesco Casella² Martin Sjölund¹ Rüdiger Franke³

¹PELAB - Programming Environments Lab, Dept. of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, {adrian.pop, per.ostlund, martin.sjолund}@liu.se

²Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milano, Italy, francesco.casella@polimi.it

³ABB, IAPG-A26, Kallstadter Str. 1, 68309 Mannheim, Germany, ruediger.franke@de.abb.com

Abstract

The equation-based object-oriented Modelica language allows easy composition of models from components. It is very easy to create very large parametrized models using component arrays of models. Current open-source and commercial Modelica tools can with ease handle models with a hundred thousand equations and a thousand states. However, when the system size goes above half a million (or more) equations the tools begin to have problems with scalability. This paper presents the new frontend of the OpenModelica compiler, designed with scalability in mind. The new OpenModelica frontend can handle much larger systems than the current one with better time and memory performance. The new frontend was validated against large models from the ScalableTestSuite library and Modelica Standard Library, with good results.

Keywords: OpenModelica, compiler, flattening, frontend, modelling, simulation, equation-based, scalability

1 Introduction and Motivation

System-level dynamic modelling and simulation is a key activity in modern system engineering design. In parallel to the detailed component design, which is performed using advanced 3D CAD, CFD and FEM software tools, system-level modelling, usually including systems of systems and large numbers of interacting components, allows predicting the dynamic performance of complex systems, which emerges from the interaction of its components.

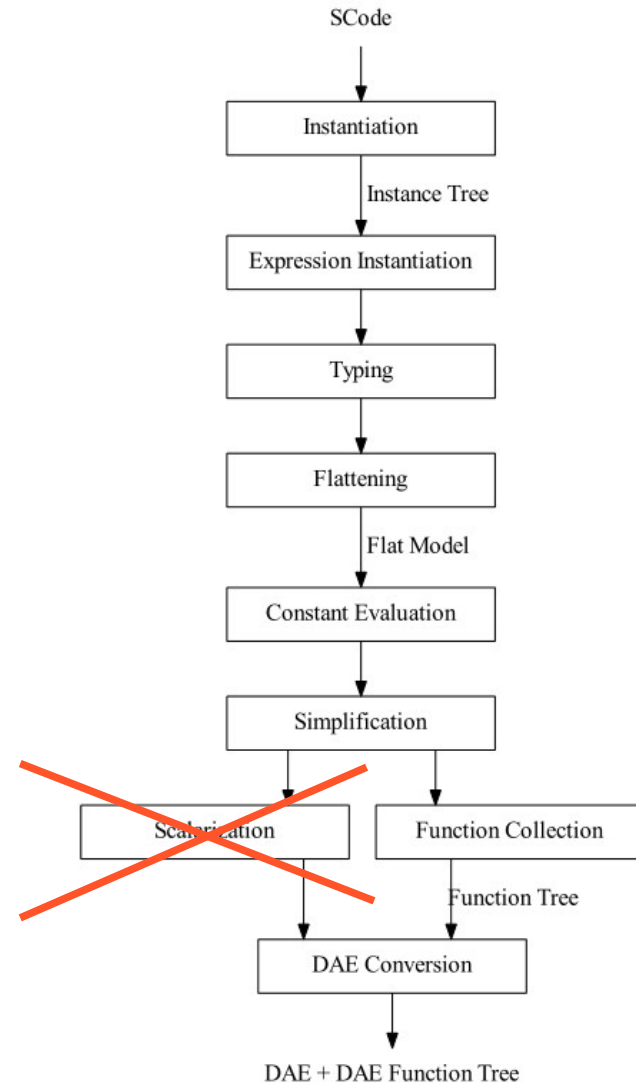
The Modelica language (Modelica Association, 2017; Fritzson, 2015) is a standardized tool-independent non-proprietary equation-based object-oriented modeling language, which was introduced 20 years ago by the non-profit Modelica Association, with strong links to industry and academia. This language, and the related eco-system of tools, model libraries and the FMI standard (Blochwitz et al., 2011), is ideally suited to system-level modeling of complex, heterogenous and multi-domain cyber-physical systems. It has become a de-facto standard in many industries, most notably the automotive one. The Modelica language is currently supported by about 10 different modeling and simulation software tools; one of them, in particular, the open-source OpenModelica software suite (Fritzson et al., 2018), is the only Modelica tool owned

and maintained by a non-profit organization – the Open Source Modelica Consortium (OSMC).

The main applications of Modelica tools so far have been the study of individual systems, such as a car's drivetrain and active suspension and steering control system, a single industrial robot, a single power plant, a single HVDC power link, the air conditioning system of a car, etc. Existing Modelica tools employ strategies and algorithms that are optimized for such system models, whose typical complexity lies in the range of 1000-50000 equations and up to a few thousand state variables. The advent of the internet-of-things paradigm is now fostering the development of innovative very large-scale cyber-physical systems, for example smart grids, or fleets of autonomous vehicles. It is also sparking a renewed interest at the modernization of traditional large-scale systems. A first example is continental-size high-voltage power generation and transmission, which is facing increasing challenges due to the introduction of power electronics equipment and to the increased penetration of intermittent renewable energy sources. A second example is district heating, possibly integrated with heat pumps and distributed power generation in an integrated electrical and thermal smart grid. See (Casella, 2015) for further examples and motivation.

Unfortunately, when Modelica is used to tackle the modelling of large-scale systems with sizes exceeding the ones mentioned above, currently available simulation software that support Modelica fall short at providing adequate performance. The time required to compile the models vastly exceeds what end users typically expect for system level studies, i.e., a few minutes at most. The size of the generated code and the memory requirements for compilers vastly exceed what is normally available on laptops and workstations used for daily work (8-16 GB).

In the last couple of years there have been some pioneering attempts at pushing the boundary of the size of Modelica models that can be handled with reasonable time and effort. In particular, some of our published papers have demonstrated the feasibility of Modelica models of high-voltage power generation and transmission systems (Braun et al., 2017; Casella et al., 2017) and of detailed models of key system components of future nuclear fusion reactors, see (Froio et al., 2017). The size of the largest models handled so far is about 750000 equations, which



Array-Preserving Flattening

- All frontend processing done keeping arrays as first class citizens
- Arrays of models are turned into (multi-dimensional) array equations
- Orders of magnitude faster if repeated objects are collected in arrays

Array-Preserving Flattening

- All frontend processing done keeping arrays as first class citizens
- Arrays of models are turned into (multi-dimensional) array equations
- Orders of magnitude faster if repeated objects are collected in arrays
- Sets of individual instances of the same model with the same structure of modifiers can be automatically collected into arrays

Non-Scalarized Flat Modelica: Example 1

```
model A
  input Real u;
  parameter Real p = 1;
  Real x;
equation
  x = p * u;
end A;
```

Non-Scalarized Flat Modelica: Example 1

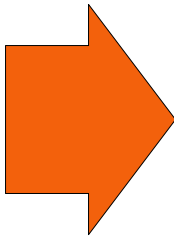
```
model A
  input Real u;
  parameter Real p = 1;
  Real x;
equation
  x = p * u;
end A;
```

```
model C
  parameter Integer N = 3;
  parameter Real p[N] = {1.0, 1.5, 2.0};
  A a[N](p = p);
equation
  a[1].u = time;
  for i in 2:N loop
    a[i].u = a[i - 1].x;
  end for;
end C;
```

Non-Scalarized Flat Modelica: Example 1

```
model A
  input Real u;
  parameter Real p = 1;
  Real x;
equation
  x = p * u;
end A;
```

```
model C
  parameter Integer N = 3;
  parameter Real p[N] = {1.0, 1.5, 2.0};
  A a[N](p = p);
equation
  a[1].u = time;
  for i in 2:N loop
    a[i].u = a[i - 1].x;
  end for;
end C;
```



```
class 'C'
  parameter Integer 'N' = 3;
  parameter Real[3] 'p' = {1.0, 1.5, 2.0};
  Real[3] 'a.x';
  parameter Real[3] 'a.p' = 'p'[:];
  Real[3] 'a.u';
equation
  for '$i1' in 1:3 loop
    'a.x'['$i1'] = 'a.p'['$i1'] * 'a.u'['$i1'];
  end for;

  'a.u'[1] = time;

  for 'i' in 2:3 loop
    'a.u'['i'] = 'a.x'['i' - 1];
  end for;
end 'C';
```

Non-Scalarized Flat Modelica: Example 2

```
model B
  parameter Integer N = 3;
  parameter Real p = 1;
  Real x[N];
  input Real u;
equation
  x[1] = u;
  for i in 2:N loop
    x[i] = x[i - 1] + p;
  end for;
```

Non-Scalarized Flat Modelica: Example 2

```
model B
  parameter Integer N = 3;
  parameter Real p = 1;
  Real x[N];
  input Real u;
equation
  x[1] = u;
  for i in 2:N loop
    x[i] = x[i - 1] + p;
  end for;
```

```
model D
  parameter Integer N = 3;
  parameter Real p[N] = {1.0, 1.5, 2.0};
  B b[N](p = p, each N = 4);
equation
  b[1].u = time;
  for i in 2:N loop
    b[i].u = b[i - 1].x[end];
  end for;
end D;
```

Non-Scalarized Flat Modelica: Example 2

```
model B
  parameter Integer N = 3;
  parameter Real p = 1;
  Real x[N];
  input Real u;
equation
  x[1] = u;
  for i in 2:N loop
    x[i] = x[i - 1] + p;
  end for;
```

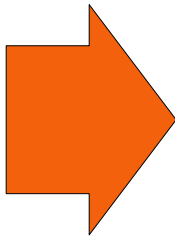
```
model D
  parameter Integer N = 3;
  parameter Real p[N] = {1.0, 1.5, 2.0};
  B b[N](p = p, each N = 4);
equation
  b[1].u = time;
  for i in 2:N loop
    b[i].u = b[i - 1].x[end];
  end for;
end D;
```

```
class 'D'
  parameter Integer 'N' = 3;
  parameter Real[3] 'p' = {1.0, 1.5, 2.0};
  Real[3] 'b.u';
  Real[3, 4] 'b.x';
  parameter Real[3] 'b.p' = 'p'[:];
  parameter Integer[3] 'b.N' = 4;
equation
  'b.x'[:,1] = 'b.u'[:];

  for '$i1' in 1:3 loop
    for 'i' in 2:4 loop
      'b.x' ['$i1', 'i'] = 'b.x' ['$i1', 'i' - 1] + 'b.p' ['$i1'];
    end for;
  end for;

  'b.u'[1] = time;

  for 'i' in 2:3 loop
    'b.u' ['i'] = 'b.x' ['i' - 1, 4];
  end for;
end 'D';
```



Non-Scalarized Flat Modelica: Example 3

```
model A
  parameter Real p = 1;
  input Real u;
  Real y;
  Real x;
equation
  der(x) = -p*x+u;
  y = 2*p*x;
end A;
```

Non-Scalarized Flat Modelica: Example 3

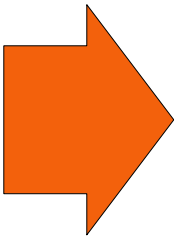
```
model A
  parameter Real p = 1;
  input Real u;
  Real y;
  Real x;
equation
  der(x) = -p*x+u;
  y = 2*p*x;
end A;
```

```
model B
  parameter Real q = 1;
  input Real u;
  output Real y;
  A aa(p = q*2);
  A ab(p = q*3);
equation
  aa.u = u;
  ab.u = aa.y;
  y = ab.y;
end B;
```

Non-Scalarized Flat Modelica: Example 3

```
model A
  parameter Real p = 1;
  input Real u;
  Real y;
  Real x;
equation
  der(x) = -p*x+u;
  y = 2*p*x;
end A;
```

```
model B
  parameter Real q = 1;
  input Real u;
  output Real y;
  A aa(p = q*2);
  A ab(p = q*3);
equation
  aa.u = u;
  ab.u = aa.y;
  y = ab.y;
end B;
```



```
class 'B'
  parameter Real 'q' = 1.0;
  input Real 'u';
  output Real 'y';
  Real[2] '$A1.x';
  Real[2] '$A1.y';
  Real[2] '$A1.u';
  parameter Real[2] '$A1.p' = {'q' * 2.0, 'q' * 3.0};
equation
  for '$i1' in 1:2 loop
    der('$A1.x' ['$i1']) = (-'$A1.p' ['$i1'] * '$A1.x' ['$i1']) + '$A1.u' ['$i1'];
  end for;

  for '$i1' in 1:2 loop
    '$A1.y' ['$i1'] = 2.0 * '$A1.p' ['$i1'] * '$A1.x' ['$i1'];
  end for;

  '$A1.u' [1] = 'u';
  '$A1.u' [2] = '$A1.y' [1];
  'y' = '$A1.y' [2];
end 'B';
```

Array-Preserving Structural Analysis and Optimization

- Matching, Sorting, Tearing, Index reduction, Alias elimination, CSE, Over/Underdetermined initialization handling, Jacobian colouring, Symbolic Jacobians

Array-Preserving Structural Analysis and Optimization

- Matching, Sorting, Tearing, Index reduction, Alias elimination, CSE, Over/Underdetermined initialization handling, Jacobian colouring, Symbolic Jacobians
- Classic E-V graph
 - E-node \leftrightarrow scalar equation
 - V-node \leftrightarrow scalar variable
- Array-Preserving E-V graph
 - E-node \leftrightarrow array equation
 - V-node \leftrightarrow array variable

Array-Preserving Structural Analysis and Optimization

- Matching, Sorting, Tearing, Index reduction, Alias elimination, CSE, Over/Underdetermined initialization handling, Jacobian colouring, Symbolic Jacobians
- Classic E-V graph
 - E-node \leftrightarrow scalar equation
 - V-node \leftrightarrow scalar variable
- Array-Preserving E-V graph
 - E-node \leftrightarrow array equation
 - V-node \leftrightarrow array variable
- Whole variable arrays in general not matched to whole equation arrays, need to consider slices, e.g. $x[1:\text{end}-1]$ or $x[1:2:\text{end}]$ or $x[\text{end}:-1:1]$
- Minimal-size array preserving is an NP-complete problem!

Array-Preserving Structural Analysis and Optimization

- Research group 1: FH Bielefeld, Germany
 - OMC New Backend development
 - K. Abdelhak, A. Heuermann, P. Hannebohm, B. Bachmann

Array-Preserving Structural Analysis and Optimization

- Research group 1: FH Bielefeld, Germany
 - OMC New Backend development
 - K. Abdelhak, A. Heuermann, P. Hannebohm, B. Bachmann

- Research group 2: University of Rosario, Argentina
 - Set-based Graph Theory
 - E. Kofman, J. F. Fernandez, D. Marzorati

Array-Preserving Structural Analysis and Optimization

- Research group 1: FH Bielefeld, Germany
 - OMC New Backend development
 - K. Abdelhak, A. Heuermann, P. Hannebohm, B. Bachmann
- Research group 2: University of Rosario, Argentina
 - Set-based Graph Theory
 - E. Kofman, J. F. Fernandez, D. Marzorati
- Research group 3: Politecnico di Milano, Italy
 - Array Graph Theory
 - M. Fioravanti, D. Cattaneo, F. Terraneo, S. Seva, S. Cherubin, G. Agosta, F., A. Leva, M. Scuttari

Current Status With OpenModelica

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)
- Scalarization

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)
- Scalarization
- Matching, sorting, index reduction, initial equations, Jacobians

Status with OpenModelica New Backend & Codegen

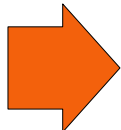
- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)
- Scalarization
- Matching, sorting, index reduction, initial equations, Jacobians
- Solved equations collected again into arrays

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)
- Scalarization
- Matching, sorting, index reduction, initial equations, Jacobians
- Solved equations collected again into arrays
- Code generation from arrays
 - Much faster code generation
 - Much faster C-code compilation
 - Much leaner generated code
 - Codegen time and code size $O(1)$ instead of $O(N)$ except for matching, sorting, index reduction, initial equations, Jacobians

Status with OpenModelica New Backend & Codegen

- Input: flattened, array-preserving Modelica AST
- Array-based pre-optimizations (alias elim., CSE, function inlining, etc.)
- Scalarization
- Matching, sorting, index reduction, initial equations, Jacobians
- Solved equations collected again into arrays
- Code generation from arrays
 - Much faster code generation
 - Much faster C-code compilation
 - Much leaner generated code
 - Codegen time and code size $O(1)$ instead of $O(N)$ except for matching, sorting, index reduction, initial equations, Jacobians



Talk later today on New Backend

ScalableTestSuite Library

- Developed since 2015 to test the performance of Modelica tools on models of increasing size
- Test models stress various aspects of the compiler

ScalableTestSuite Library

- Developed since 2015 to test the performance of Modelica tools on models of increasing size
- Test models stress various aspects of the compiler
- Current performance with [NF and OB](#)
- Current performance with [NF and NB](#)
- Current performance of larger models with [NF and NB](#)

Example 1: CascadedFirstOrder

```
model CascadedFirstOrder
  parameter Integer N = 10 "Order of the system";
  parameter Modelica.Units.SI.Time T=1 "System delay";
  final parameter Modelica.Units.SI.Time tau=T/N;
  Real x[N] ( each start = 0, each fixed = true);
equation
  tau*der(x[1]) = 1 - x[1];
  for i in 2:N loop
    tau*der(x[i]) = x[i-1] - x[i];
  end for;
end CascadedFirstOrder;
```

Example 1: CascadedFirstOrder

```
model CascadedFirstOrder
  parameter Integer N = 10 "Order of the system";
  parameter Modelica.Units.SI.Time T=1 "System delay";
  final parameter Modelica.Units.SI.Time tau=T/N;
  Real x[N] ( each start = 0, each fixed = true);
equation
  tau*der(x[1]) = 1 - x[1];
  for i in 2:N loop
    tau*der(x[i]) = x[i-1] - x[i];
  end for;
end CascadedFirstOrder;
```

[.CascadedFirstOrder N 100 \(sim\)](#)
[.CascadedFirstOrder N 200 \(sim\)](#)
[.CascadedFirstOrder N 400 \(sim\)](#)
[.CascadedFirstOrder N 800 \(sim\)](#)
[.CascadedFirstOrder N 1600 \(sim\)](#)
[.CascadedFirstOrder N 3200 \(sim\)](#)
[.CascadedFirstOrder N 6400 \(sim\)](#)
[.CascadedFirstOrder N 12800 \(sim\)](#)
[.CascadedFirstOrder N 25600 \(sim\)](#)

Old Backend		New Backend	
Simulate	Total buildModel	Simulate	Total buildModel
0.03	1.56	0.02	1.31
0.04	1.67	0.03	1.31
0.06	1.97	0.05	1.31
0.13	2.06	0.09	1.35
0.34	3.23	0.23	1.72
0.90	4.95	0.66	1.80
2.72	8.59	2.07	2.45
8.64	17.27	7.15	3.32
34.97	39.55	39.40	5.52

Example 1: CascadedFirstOrder

```

model CascadedFirstOrder
  parameter Integer N = 10 "Order of the system";
  parameter Modelica.Units.SI.Time T=1 "System delay";
  final parameter Modelica.Units.SI.Time tau=T/N;
  Real x[N] ( each start = 0, each fixed = true);
equation
  tau*der(x[1]) = 1 - x[1];
  for i in 2:N loop
    tau*der(x[i]) = x[i-1] - x[i];
  end for;
end CascadedFirstOrder;

```

	Old Backend		New Backend	
	Simulate	Total buildModel	Simulate	Total buildModel
.CascadedFirstOrder N 100 (sim)	0.03	1.56	0.02	1.31
.CascadedFirstOrder N 200 (sim)	0.04	1.67	0.03	1.31
.CascadedFirstOrder N 400 (sim)	0.06	1.97	0.05	1.31
.CascadedFirstOrder N 800 (sim)	0.13	2.06	0.09	1.35
.CascadedFirstOrder N 1600 (sim)	0.34	3.23	0.23	1.72
.CascadedFirstOrder N 3200 (sim)	0.90	4.95	0.66	1.80
.CascadedFirstOrder N 6400 (sim)	2.72	8.59	2.07	2.45
.CascadedFirstOrder N 12800 (sim)	8.64	17.27	7.15	3.32
.CascadedFirstOrder N 25600 (sim)	34.97	39.55	39.40	5.52
CascadedFirstOrder N 12800 (sim)			30.44	3.08
CascadedFirstOrder N 25600 (sim)			86.27	4.85
CascadedFirstOrder N 51200 (sim)			246.40	8.97
CascadedFirstOrder N 102400 (sim)			301.53	17.43
CascadedFirstOrder N 204800 (sim)			302.01	37.59
CascadedFirstOrder N 409600 (sim)			302.01	86.27

Largest model: 400.000 equations, 400.000 states

Example 2: HarmonicOscillator

```
model HarmonicOscillator
  import SIunits = Modelica.Units.SI;
  parameter Integer N = 2 "Number of masses in the system";
  parameter SIunits.Mass m = 1 "Mass of each node";
  parameter SIunits.TranslationalSpringConstant k = 10;
  SIunits.Position x[N] "Positions of the masses";
  SIunits.Velocity v[N] "Velocity of the masses";
equation
  for i in 1:N loop
    der(x[i]) = v[i];
  end for;
  m*der(v[1]) = k*(x[2]-x[1]);
  for i in 2:N - 1 loop
    m*der(v[i]) = k*(x[i-1] - x[i]) + k*(x[i+1] - x[i]);
  end for;
  m*der(v[N]) = k*(x[N-1]-x[N]);
initial equation
  x[1] = N;
  v[1] = 0;
  for i in 2:N loop
    x[i] = 0;
    v[i] = 0;
  end for;
end HarmonicOscillator;
```

Example 2: HarmonicOscillator

	Old Backend		New Backend	
	Simulate	Total buildModel	Simulate	Total buildModel
HarmonicOscillator N 100 (sim)	0.05	1.78	0.04	1.27
HarmonicOscillator N 200 (sim)	0.09	1.79	0.07	1.36
HarmonicOscillator N 400 (sim)	0.18	2.18	0.14	1.40
HarmonicOscillator N 800 (sim)	0.48	3.16	0.40	1.77
HarmonicOscillator N 1600 (sim)	1.29	5.01	1.09	1.89
HarmonicOscillator N 3200 (sim)	6.56	8.56	6.62	2.39

Example 2: HarmonicOscillator

	Old Backend		New Backend	
	Simulate	Total buildModel	Simulate	Total buildModel
HarmonicOscillator N 100 (sim)	0.05	1.78	0.04	1.27
HarmonicOscillator N 200 (sim)	0.09	1.79	0.07	1.36
HarmonicOscillator N 400 (sim)	0.18	2.18	0.14	1.40
HarmonicOscillator N 800 (sim)	0.48	3.16	0.40	1.77
HarmonicOscillator N 1600 (sim)	1.29	5.01	1.09	1.89
HarmonicOscillator N 3200 (sim)	6.56	8.56	6.62	2.39
HarmonicOscillator N 3200 (sim)			2.60	2.19
HarmonicOscillator N 6400 (sim)			8.46	2.97
HarmonicOscillator N 12800 (sim)			23.86	5.06
HarmonicOscillator N 25600 (sim)			98.79	9.15

Largest model: 50.000 equations, 50.000 states

Example 3: Countercurrent Heat Exchanger

```
model CounterCurrentHeatExchangerEquations
  import SIunits = Modelica.Units.SI;
  parameter SIunits.Length L;
  parameter Integer N = 2;
  parameter SIunits.MassFlowRate wB;
  parameter SIunits.Area areaA;
  parameter SIunits.Area areaB;
  parameter SIunits.Density rhoA;
  parameter SIunits.Density rhoB;
  parameter SIunits.SpecificHeatCapacity cpA;
  parameter SIunits.SpecificHeatCapacity cpB;
  parameter SIunits.SpecificHeatCapacity cpW;
  parameter SIunits.CoefficientOfHeatTransfer gammaA;
  parameter SIunits.CoefficientOfHeatTransfer gammaB;
  parameter SIunits.Length omega;
  final parameter SIunits.Length l = L / (N - 1)
  SIunits.MassFlowRate wA;
  SIunits.HeatFlowRate QA[N - 1];
  SIunits.HeatFlowRate QB[N - 1];
  SIunits.Temperature TA[N];
  SIunits.Temperature TB[N];
  SIunits.Temperature TAtilde[N - 1];
  SIunits.Temperature TBtilde[N - 1];
  SIunits.Temperature TW[N - 1];
  SIunits.HeatFlowRate QtotA;
  SIunits.HeatFlowRate QtotB;

  initial equation
    for i in 1:N - 1 loop
      TAtilde[i] = 300;
      TBtilde[i] = 300;
      TW[i] = 300;
    end for;
  equation
    TA[1] = if time < 8 then 300 else 301;
    for i in 2:N loop
      TA[i] = TAtilde[i - 1];
    end for;
    TB[N] = 310;
    for i in 1:N - 1 loop
      TB[i] = TBtilde[i];
    end for;
    wA = if time < 15 then 1 else 1.1;
    for i in 1:N - 1 loop
      rhoA * l * areaA * cpA * der(TAtilde[i]) =
        wA * cpA * TA[i] - wA * cpA * TA[i + 1] + QA[i];
      rhoB * l * areaB * cpB * der(TBtilde[N - i]) =
        wB * cpB * TB[N - i + 1] - wB * cpB * TB[N - i] - QB[N - i];
      QA[i] = (TW[i] - (TA[i] + TA[i + 1]) / 2) * gammaA * omega * l;
      QB[N - i] = ((TB[N - i + 1] + TB[N - i]) / 2 - TW[N - i]) * gammaB * omega * l;
      cpW / (N - 1) * der(TW[i]) = (-QA[i]) + QB[i];
    end for;
    QtotA = sum(QA);
    QtotB = sum(QB);
end CounterCurrentHeatExchangerEquations;
```

Example 3: Countercurrent Heat Exchanger

[CounterCurrentHeatExchangerEquations N 10 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 20 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 40 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 80 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 160 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 320 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 640 \(sim\)](#)
[CounterCurrentHeatExchangerEquations N 1280 \(sim\)](#)

Old Backend

Simulate Total buildModel

0.03	1.64
0.03	1.62
0.05	1.92
0.09	2.08
0.20	3.03
0.49	4.42
1.16	7.14
2.99	15.00

New Backend

Simulate Total buildModel

0.02	1.25
0.03	1.36
0.04	1.35
0.06	1.41
0.11	1.35
0.28	1.77
0.79	2.33
2.69	2.94

Example 3: Countercurrent Heat Exchanger

	Old Backend		New Backend	
	Simulate	Total buildModel	Simulate	Total buildModel
CounterCurrentHeatExchangerEquations N 10 (sim)	0.03	1.64	0.02	1.25
CounterCurrentHeatExchangerEquations N 20 (sim)	0.03	1.62	0.03	1.36
CounterCurrentHeatExchangerEquations N 40 (sim)	0.05	1.92	0.04	1.35
CounterCurrentHeatExchangerEquations N 80 (sim)	0.09	2.08	0.06	1.41
CounterCurrentHeatExchangerEquations N 160 (sim)	0.20	3.03	0.11	1.35
CounterCurrentHeatExchangerEquations N 320 (sim)	0.49	4.42	0.28	1.77
CounterCurrentHeatExchangerEquations N 640 (sim)	1.16	7.14	0.79	2.33
CounterCurrentHeatExchangerEquations N 1280 (sim)	2.99	15.00	2.69	2.94
CounterCurrentHeatExchangerEquations N 1280 (sim)			7.84	2.43
CounterCurrentHeatExchangerEquations N 2560 (sim)			20.33	4.50
CounterCurrentHeatExchangerEquations N 5120 (sim)			54.78	9.60
CounterCurrentHeatExchangerEquations N 10240 (sim)			148.39	24.46

Largest model: 70.000 equations, 30.000 states

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag
- Very small flattening time, new frontend complete
- Drastically reduced code generation and compilation time
- Drastically reduced generated code size

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag
- Very small flattening time, new frontend complete
- Drastically reduced code generation and compilation time
- Drastically reduced generated code size
- Some parts of structural analysis still carried out on scalarized model
Expected performance $O(N)$, some work yet to be done

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag
- Very small flattening time, new frontend complete
- Drastically reduced code generation and compilation time
- Drastically reduced generated code size
- Some parts of structural analysis still carried out on scalarized model
Expected performance $O(N)$, some work yet to be done
- Sparse algebraic solvers don't work yet (too bad...)

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag
- Very small flattening time, new frontend complete
- Drastically reduced code generation and compilation time
- Drastically reduced generated code size
- Some parts of structural analysis still carried out on scalarized model
Expected performance $O(N)$, some work yet to be done
- Sparse algebraic solvers don't work yet (too bad...)
- $O(1)$ set-graph algorithms could be used for structural analysis, further improving performance. Currently evaluating with University of Rosario research group.

Summary: OMC with New FrontEnd & New Backend

- Some interesting results already available with latest nightly build, just set `--newBackend` translation flag
- Very small flattening time, new frontend complete
- Drastically reduced code generation and compilation time
- Drastically reduced generated code size
- Some parts of structural analysis still carried out on scalarized model
Expected performance $O(N)$, some work yet to be done
- Sparse algebraic solvers don't work yet (too bad...)
- $O(1)$ set-graph algorithms could be used for structural analysis, further improving performance. Currently evaluating with University of Rosario research group.
- Still very early stage, needs 2-3 more years of development to reach maturity
- Some classes of models (e.g. large power grid models) expected to work by the end of 2023

Current Status with MARCO

The MARCO Compiler

- MARCO (**M**odelica **A**dvanced **R**esearch **C**ompiler) is under development at Politecnico di Milano since 2019
- Goal: provide fastest possible compilation and simulation for a selected subset of Modelica models

The MARCO Compiler

- MARCO (**M**odelica **A**dvanced **R**esearch **C**ompiler) is under development at Politecnico di Milano since 2019
- Goal: provide fastest possible compilation and simulation for a selected subset of Modelica models
- Main features:
 - Written in C++
 - Based on LLVM infrastructure (the same of clang)
 - Input: Flat non-scalarized Modelica code from OMC
 - Generates LLVM-IR code (instead of C): more room for optimization
 - No runtime garbage collection, statically allocated memory

The MARCO Compiler

- MARCO (**M**odelica **A**dvanced **R**esearch **C**ompiler) is under development at Politecnico di Milano since 2019
- Goal: provide fastest possible compilation and simulation for a selected subset of Modelica models
- Main features:
 - Written in C++
 - Based on LLVM infrastructure (the same of clang)
 - Input: Flat non-scalarized Modelica code from OMC
 - Generates LLVM-IR code (instead of C): more room for optimization
 - No runtime garbage collection, statically allocated memory
- Currently supports:
 - Continuous-time models
 - Records, arrays, functions
 - Explicit Euler with closed-form solution of strong components
 - daeMode integration with IDA

Example: ThermalChipOO

```
model Volume
  parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon";
  parameter Types.Density rho = 2329 "Density of silicon";
  parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon";
  parameter Types.Temperature Tstart = 273.15 + 40;
  parameter Types.ThermalCapacitance C "Thermal capacitance of a volume";
  parameter Types.ThermalConductance Gx "Thermal conductance of half a volume, x direction";
  parameter Types.ThermalConductance Gy "Thermal conductance of half a volume, y direction";
  parameter Types.ThermalConductance Gz "Thermal conductance of half a volume, z direction";

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

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

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

model PowerSource
  Interfaces.HeatPort port;
  Types.PowerInput Q;
equation
  port.Q = -Q;
end PowerSource;

model TemperatureSource
  Interfaces.HeatPort port;
  Types.Temperature T = 298.15;
equation
  port.T = T;
end TemperatureSource;
```

Example: ThermalChipOO

```
partial model BaseThermalChip
  parameter Integer N = 4 "Number of volumes in the x direction";
  parameter Integer M = 4 "Number of volumes in the y direction";
  parameter Integer P = 4 "Number of volumes in the z direction";
  parameter Types.Length L = 12e-3 "Chip length in the x direction" annotation( ... );
  parameter Types.Length W = 12e-3 "Chip width in the y direction" annotation( ... );
  parameter Types.Length H = 4e-3 "Chip height in the z direction" annotation( ... );
  parameter Types.ThermalConductivity lambda = 148 "Thermal conductivity of silicon" annotation( ... );
  parameter Types.Density rho = 2329 "Density of silicon" annotation( ... );
  parameter Types.SpecificHeatCapacity c = 700 "Specific heat capacity of silicon" annotation( ... );
  parameter Types.Temperature Tstart = 273.15 + 40;
  final parameter Types.Length l = L / N "Chip length in the x direction";
  final parameter Types.Length w = W / M "Chip width in the y direction";
  final parameter Types.Length h = H / P "Chip height in the z direction";
  parameter Types.Temperature Tt = 273.15 + 40 "Prescribed temperature of the top surface" annotation( ... );
  final parameter Types.ThermalCapacitance C = rho*c*l*w*h "Thermal capacitance of a volume";
  final parameter Types.ThermalConductance Gx = lambda*w*h/l "Thermal conductance of a volume, x direction";
  final parameter Types.ThermalConductance Gy = lambda*l*h/w "Thermal conductance of a volume, y direction";
  final parameter Types.ThermalConductance Gz = lambda*l*w/h "Thermal conductance of a volume, z direction";

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

  output Types.Temperature Tct1 = vol[1,1,1].T "Top layer corner volume temperature";
  output Types.Temperature Tct2 = vol[1,N,1].T "Top layer corner volume temperature";
  output Types.Temperature Tct3 = vol[N,N,1].T "Top layer corner volume temperature";
  output Types.Temperature Tct4 = vol[N,1,1].T "Top layer corner volume temperature";
  output Types.Temperature Tcb1 = vol[1,1,P].T "Bottom layer corner volume temperature";
  output Types.Temperature Tcb2 = vol[1,N,P].T "Bottom layer corner volume temperature";
  output Types.Temperature Tcb3 = vol[N,N,P].T "Bottom layer corner volume temperature";
  output Types.Temperature Tcb4 = vol[N,1,P].T "Bottom layer corner volume temperature";
```


Example: ThermalChipOO

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

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

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

Example: ThermalChipOO

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

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

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

model ThermalChipSimpleBoundary
  extends BaseThermalChip;
  parameter Types.Power Ptot = 100 "Total power consumption";
  final parameter Types.Power Pv = Ptot / (N * M / 2);
  PowerSource Qsource[N,div(M,2)](each Q = Pv);
equation
  connect(Qsource.port, vol[:, 1:div(M,2), P].center);
end ThermalChipSimpleBoundary;
```

Example: ThermalChipOO

- Simulated transient: response to applied thermal power on half of the active surface, explicit fixed-time step Euler
- CPU: i9-12900KF
- OS: Ubuntu 20.04 LTS

Example: ThermalChipOO

- Simulated transient: response to applied thermal power on half of the active surface, explicit fixed-time step Euler
- CPU: i9-12900KF
- OS: Ubuntu 20.04 LTS

Name	Vars	States	Compile Time OMC	Compile Time MARCO	Run Time OMC	RunTime Marco
ThermalChipSimpleBoundaryOO N=4, M=4, P=4	~1k	64	1.367s	0.390s	0.063s	0.012s
ThermalChipSimpleBoundaryOO N=40, M=40, P=40	~1M	~64k	OOM after 5m 34.814s	0.492s	N/A	2.696s
ThermalChipSimpleBoundaryOO N=100, M=100, P=100	~16M	~1M	N/A	1.108s	N/A	28.640s

Conclusions & Outlook

Conclusions

- Systems-of-systems modelling can play a crucial role supporting the design and deployment of innovative distributed cyber-physical systems.
- Modelica is ideally suited for this task: high-level, declarative, modular.

Conclusions

- Systems-of-systems modelling can play a crucial role supporting the design and deployment of innovative distributed cyber-physical systems.
- Modelica is ideally suited for this task: high-level, declarative, modular.
- Modelica compiler technology needs a quantum leap to support array-preserving code generation to support these applications.

Conclusions

- Systems-of-systems modelling can play a crucial role supporting the design and deployment of innovative distributed cyber-physical systems.
- Modelica is ideally suited for this task: high-level, declarative, modular.
- Modelica compiler technology needs a quantum leap to support array-preserving code generation to support these applications.
- OpenModelica development is heading in this direction since 2015.
- The New Backend will play a pivotal role in making OMC usable in this area.

Conclusions

- Systems-of-systems modelling can play a crucial role supporting the design and deployment of innovative distributed cyber-physical systems.
- Modelica is ideally suited for this task: high-level, declarative, modular.
- Modelica compiler technology needs a quantum leap to support array-preserving code generation to support these applications.
- OpenModelica development is heading in this direction since 2015.
- The New Backend will play a pivotal role in making OMC usable in this area.
- Early results seem very promising, more will come in 2023, probably 2-3 years until maturity

Conclusions

- Systems-of-systems modelling can play a crucial role supporting the design and deployment of innovative distributed cyber-physical systems.
- Modelica is ideally suited for this task: high-level, declarative, modular.
- Modelica compiler technology needs a quantum leap to support array-preserving code generation to support these applications.
- OpenModelica development is heading in this direction since 2015.
- The New Backend will play a pivotal role in making OMC usable in this area.
- Early results seem very promising, more will come in 2023, probably 2-3 years until maturity
- Other tools such as MARCO can also benefit from OMC technology, advancing in this area with somewhat different perspective and goals

**Thank you for your
kind attention!**